Introduction to Machine Learning

From Math to Code

Ruye Wang

Contents

Preface *page* ix

Part I Mathematical Fundations 1

1 Solving Equations 5
1.1 Linear Equation Systems 6
1.2 The Bisection and Secant Methods 8
1.2.1 The Bisection Search 8
1.2.2 The Secant Method 10
1.3 Fixed-point Iteration 11
1.4 Newton-Raphson Method (Univariate) 22
1.5 Newton-Raphson Method (Multivariate) 26
Problem 33
1.A Order of Convergence of the Secant Method 34
1.B Order of Convergence of the Newton-Raphson Method 35
1.C Proofs of the Fixed Point and Contraction Mapping Theorems 36

2 Unconstrained Optimization 38
2.1 Optimization by Solving Equations 39
2.2 Newton's Method 40
2.3 Gradient Descent Method 46
2.4 Line minimization 49
2.5 Quasi-Newton Methods 56
2.6 Conjugate gradient method 63
2.7 Issues of Local/Global Minimum 76
Problems 77

3 Constrained Optimization 79
3.1 Optimization with Equality Constraints 80
3.2 Optimization with Inequality Constraints 83
3.3 Duality and KKT Conditions 88
3.4 Linear Programming (LP) 92
3.5 The Simplex Algorithm 98
3.6 Quadratic Programming (QP) 104

3.7 Interior Point Methods 106
Problems 114

Part II Regression 117

4 Bias-Variance Tradeoff and Overfitting vs Underfitting 121
4.1 Supervised Learning as Optimization 121
4.2 Bias-Variance Tradeoff 125
4.3 Cross-Validation 129
4.4 Regularization and Ensemble Learning 130

5 Linear Regression 132
5.1 Linear Least Squares (LLS) Regression 132
5.2 Ridge Regression 143
5.3 Regression Based on Basis Functions 146
5.4 Bayesian Regression 154
Problems 162

6 Nonlinear Regression 165
6.1 Nonlinear Least Squares Regression 165
6.2 Parameter Estimation by Natural Gradient Descent 172
Problems 173

7 Logistic and Softmax Regression 174
7.1 Logistic Regression and Binary Classification 174
7.2 Softmax Regression for Multiclass Classification 182
Problems 190

8 Gaussian Process Regression and Classification 192
8.1 Gaussian Process Regression 192
8.2 Gaussian Process Classifier – Binary 198
8.3 Gaussian Process Classifier – Multi-Class 205
Problems 214

Part III Feature Extraction 217

9 Feature Selection 220
9.1 Distances and Separability Measurements 220
Problems 224

10 Principal Component Analysis 227
10.1 Covariance and Correlation 227
10.2 Karhunen-Loève Transformation 230
10.3 Optimality of KLT 232

10.4 Geometric Interpretation of the KLT 235
10.5 Computation of the KLT 237
10.6 Comparison with Other Orthogonal Transforms 239
10.7 Application to Image Data 241
10.8 PCA for Feature Extraction 247
Problems 251

11 Variations of PCA 254
11.1 Kernel Methods 254
11.2 Kernel PCA 258
11.3 Factor Analysis and Expectation Maximization 261
11.4 Probabilistic PCA 268
11.5 Classical Multidimensional Scaling 273
11.6 t-Distributed Stochastic Neighbor Embedding 278
Problems 285

12 Independent Component Analysis 287
12.1 Independence and Non-Gaussianity 287
12.2 Preprocessing and Whitening 289
12.3 Non-Gaussianity and FastICA 290
12.4 Likelihood and Independence Maximization 294
Problems 300

Part IV Classification 301

13 Statistic Classification 305
13.1 Discriminative vs. Generative Methods for Classification 305
13.2 K Nearest Neighbors and Minimum Distance Classifiers 307
13.3 Naive Bayes Classification 309
13.4 Adaptive Boosting 321
Problems 328

14 Support Vector machine 334
14.1 Maximum Margin and Support Vectors 334
14.2 Kernel SVM 343
14.3 Soft Margin SVM 344
14.4 Sequential Minimal Optimization Algorithm 347
14.5 Multiclass Classification 356
14.6 Kernelized Bayes classifier 362
Problems 369

15 Clustering Analysis 371
15.1 K-means clustering 371
15.2 Gaussian mixture model 377

15.3 Bernoulli Mixture Model 387
Problems 392

16 Hierarchical Classifiers 394
16.1 Bottom-Up vs Top-Down Methods 394
16.2 Binary Hierarchical Classification 396
16.3 Binary Hierarchical Clustering 401
Problems 407

Part V Neural Networks 409

17 Biologically Inspired Networks 413
17.1 Biological Neural Network Modeling 413
17.2 Hebbian Learning 417
17.3 Hopfield Network 420

18 Perceptron-Based Networks 424
18.1 Perceptron 424
18.2 Back Propagation 433
18.3 Autoencoder 443
18.4 Deep Learning 451
Problems 457

19 Competition-Based Networks 459
19.1 Competitive Learning Network 459
19.2 Self-Organizing Map (SOM) 464
Problems 472

Part VI Reinforcement Learning 475

20 Introduction to Reinforcement Learning 479
20.1 Markov Decision Process 479
20.1.1 Markov Chain 479
20.1.2 Markov Reward Process 480
20.1.3 Markov Decision Process 483
20.2 Model-Based Planning 486
20.3 Model-Free Evaluation and Control 491
20.3.1 Monte Carlo (MC) Algorithms 496
20.3.2 Temporal Difference (TD) Algorithms 499
20.3.3 TD(λ) Algorithm 506
20.4 Value Function Approximation 511
20.5 Control Based on Function Approximation 516
20.6 Deep Q-learning 518
20.7 Policy Gradient Methods 522

Problems 529

Part VII Large Language Models 531

21 Large Language Models 535
21.1 Natural Language Processing and Modeling 535
21.2 The Transformer Architecture 537
21.3 Attention Is All You Need 539
21.4 Backpropagation Learning 542
21.5 Transfer Learning in NLP 547
21.6 Reinforcement Learning with Human Feedback 549
21.7 Scaling Laws 550
21.8 Mathematical Operations in the Transformer Architecture 551
21.9 Emergence 552
21.10 Do LLMs Unstand? 554
21.11 Future Directions 558
21.12 Applications of Transformers Beyond NLP 561
21.13 Generative Methods Beyond Transformers 564

Appendix A A Review of Linear Algebra 569
A.1 Inner Product Space 569
A.1.1 Vector Space 569
A.1.2 Orthogonal Basis and Gram-Schmidt Process 576
A.2 Matrices 578
A.2.1 Rank, Trace, Determinant, Transpose, and Inverse 578
A.2.2 Normal, Unitary, and Similar Matrices 580
A.2.3 Positive/Negative (Semi)-Definite Matrices 581
A.2.4 Woodbury Matrix Identity and Sherman-Morrison Formula 582
A.2.5 Inverse and Determinant of Partitioned Symmetric Matrices 583
A.2.6 Unitary Transform 585
A.3 Eigenvalues and Eigenvectors 587
A.3.1 Eigenvalue Decomposition 587
A.3.2 Generalized Eigenvalue Problem 591
A.3.3 Rayleigh Quotient 593
A.3.4 Normal Matrices and Diagonalizability 593
A.3.5 Singular Value Decomposition 595
A.4 Vector and Matrix Calculations 598
A.4.1 Vector Norms 598
A.4.2 Matrix Norms 602
A.4.3 Vector and Matrix Differentiation 610
A.5 The Fundamental Theorem of Linear Algebra 611
A.5.1 Rank-Nullity Theorem 611
A.5.2 Solving Linear Equation Systems 613
A.6 Pseudo-Inverse 625

A.6.1 Pseudo-Inverse 625
A.6.2 Pseudo-Inverse Solutions Based on SVD 627
A.7 Taylor Series Expansion 634
A.8 Miscellaneous 636
A.8.1 Centering Matrix 636
A.8.2 Normal Direction of a Plane 637
A.8.3 Convexity 639
A.8.4 Jensen Inequality 640
A.8.5 Sensitivity and Conditioning 641
A.8.6 Some Useful Inequalities 650

Appendix B A Review of Probability and Statistics 653

Notes 697

Preface

Machine learning (ML), as a branch of artificial intelligence (AI), has reached its maturity, due to the advancements in both computing power and new theoretical and algorithmic development in the recent decades. The technology has found many new applications in a wide spectrum of areas in both industry and our daily life, thereby generating a significant amount passion among college students as well as practicing professionals with different backgrounds to gain the knowledge and master the skill in ML, as a fascinating intersection of applied mathematics, computer science, and engineering. This book is motivated by such passion among my students during my teaching of the subject over several years, and it grew out of the lecture notes developed for the course. The book is written for upper-level undergraduate or first year graduate students in computer science, engineering, or any other relevant majors, and it can also be used as a reference for practicing professionals of different background but interested in ML.

A different Pedagogical Phhilosophy

The idea of converting notes into a textbook came from the realization that my approach of teaching the subject is not quite the same as many of the existing textbooks on the subject, which can be roughly categorized into either theoretical or practical type. Books of the theoretical type treat ML as a subject of applied mathematics, emphasizing the mathematical theory as the background of the various ML methods. Such books typically provide detailed mathematical background and rigorous derivations of the algorithms, but they pay little attention to the actual implementation of the algorithms discussed. Readers of these books may be left unsatisfied if they also want to know how the algorithms are actually implemented in code and applied to real-world problems. On the other hand, books of the practical type treat the subject of ML as a computational toolbox, which is black or a grey at best, in the sense that they emphasize mostly the application of the tools in the toolbox while paying little attention to the mathematical background of the algorithms. Such books typically rely on certain off-the-shelf commercial software packages, and they discuss mostly how the functions in some built-in library are used, but very little why the underneath algorithms work (or not work under certain conditions). Some of such books may by in the style of manual books or cooking recipes, providing little more than a set of steps to follow. While such books may serve the purpose of certain readers,

others may feel unsatisfied knowing only how to use a black box blindly, when they also desire to open the box to learn its inner workings.

Different from either of these two approaches, this book is intended as neither a theoretical mathematics book nor a practical manual book, but a combination of both. It is written based on the belief that machine learning can be best learned as a subject of both applied mathematics for the theoretical background of the algorithms, and a branch of engineering for the implementation and application of the algorithms. As suggested by the subtitle *From Math to Code*, this book attempts to bridge the gap between the theoretical foundation of the ML methods and their algorithmic and code level implementation, so that readers seriously interested in the subject can learn to understand both *why* and *how* the ML algorithms work. Once they gain such insights of an algorithm, as well as its strength and weakness, conditions and limitations, they will be able to not only use the algorithm properly and effectively, but also possibly modify and improve it while solving their specific real world problems.

Pedagogical Feature and Code

In addition to the rigorous mathematical presentation needed for the in-depth understanding of the algorithms, also provided in the book are a large number of simple and thoroughly worked out examples following most of the algorithms discussed in the text to show how they are applied as well as implemented, and to illustrate their results, such as different effects due to different parameter settings. All such examples are based on home-brewed software in both Matlab and Python without using any existing software or libraries. Moreover, very often some essential segments of the Matlab implementation of an algorithm are included in the text following the mathematical background of the algorithm, so that the reader can see how a mathematical idea is converted from equations to code to be actually realized. It is the author's experience and belief that studying the code implementation of an algorithm is an effective way to learn how specifically the algorithm works. By studying the code as well as the mathematics, the reader is exposed to both languages and herefore gain the valuable experience of translating mathematical ideas into code implementations. The Matlab code is written in such a way that it matches as closely as possible to the equations in the text for readability, while leaving the computational efficiency to Python, the currently dominant language for large-scale data analysis. The Python programs associated with the chapters are provided separately at: https://github.com/njkrichardson/mlam

Regarding the Mathematics

It is the author's firm belief that to thoroughly understand the ML algorithms covered in the book, it is necessary to also learn their mathematical backgrounds, mostly in the areas of multivariable calculus, linear algebra, probability and statistics. In addition, some basic understanding of optimization (both constrained and unconstrained) and dynamic programming is needed, as many ML algorithms are essentially an optimization problem. On the other hand, the mathematical background of all potential readers of the book may span span

a wide spectrum. Some readers may not necessarily have all such background knowledge and they may likely feel difficult following the mathematics in certain parts of the book. Therefore the concern regarding the proper depth and amount of mathematics in the book needs to be addressed to satisfy most of the readers, so that the mathematics in the book serves as a useful tool that helps the readers to better understand the algorithms, instead of roadblocks that discourage some readers from studying the subject of ML.

The approach taken by this book is somewhat different from many of the ML textbooks intended to seriously cover the theoretical background of the various ML algorithms, which assume the readers are either already knowledgeable in these subjects from their prior studies, or they are willing to find proper references and learn the materials on their own. Differently, this book is developed as a self-contained textbook, so that readers unfamiliar with some of the mathematical topics can find all necessary background information as well as the main ML topics in the same volumn without the need for referencing any other books. They can find a brief summary of linear algebra and probability/statistics in Appendices A and B respectively, and also a more in-depth discussion for optimization in Part I of the main text. However, such readers do not have to study all these topics as preparatory knowledge before they start to study the ML algorithms, instead, they are encouraged to refer to these mathematical reviews only when they run into difficulties while following the derivation of specific ML algorithms due to their unfamiliarity of the background mathematics. In such a case they can take a brief digression to read a few relevant paragraphs in Part I, Appendix A or B to quickly gain the necessary background knowledge and then come back right away to continue their study of the main topic. Such a brief digression within the book would be much more convenient than looking for a proper reference book and then studying a relevant section in it.

It is possible that some readers may still feel difficult understanding the mathematics in certain parts of the book due to their lack of either the relevant knowledge or enough experience to follow mathematical derivations. Such readers are encouraged to skip such parts in the text when studying the topics the first time. The study of ML, as well as any other subject for this matter, is necessarily a repetitive process which is by no means linear. A more knowledgeable and experienced reader may want to thoroughly understand an algorithm by going through all background mathematics, but this is not necessarily the only way to learn the subject. If less experienced readers run into difficulty following the derivation (occationally lengthy and tedious) of a certain algorithm when studying it the first time, they are encouraged to skip the intermediate steps and directly jump to the final result, so that they can quickly grasp the basic idea and concept of interest, while keeping the option of coming back to study the detailed mathematics later when needed.

End of Chapter Problems – Unique Approach

Different from the typical homework problems in many ML textbooks, the homework problems in this book are all about implementation of the algorithms

covered in the chapter. The students are required to develop their own code (in Matlab, Python, or any other language of choice) to implement the algorithms and test them based on some given datasets. Such coding exercises are based on the belief that the ultimate goal for studying ML is to be able to apply various algorithms to solve real problems, and the ultimate test whether the students have truly understood an algorithm is to see if they are able to implement the algorithm by their own code based on the theoretical background of the algorithm, without relying on existing software readily available in some commercial libraries or packages. Such libraries and packages can be highly valuable for anyone to use for solving their practical problems, but they may not be the right tool if used as a blackbox for anyone trying to learn and understand the innerworkings of ML algorithms.

As the guiding philosophy of the book, such a belief is also put into practice during the development of the book. Most of the algorithms mentioned in the text were implemented by the author's own Matlab code developed from scratch, and tested on some sample datasets as shown in the large number of examples in the text. Moreover, some code segments and functions flecting the essential parts of each of the algorithms also provided in the text to serve two purposes. First, according to the author's own experience, seeing how an algorithm is actually implemented in code makes it more efficient to thorough learn the algorithm. Second, the provided code segments make the required coding homework less challenging, so that the students do not have to spend too much time to develop their code from scratch. However, a proper tradeoff needs to be made in terms of how much of code to provide. It is hoped that the amount of code in the text achieves a reasonable balance between the two extremes of providing complete code and no code at all.

Most of the homework problems ask the students to develop their own Matlab code to implement the algorithms covered in the chapters, although any other language can be used if a student so prefers. Matlab is recommended as it allows the students to concentrate on the conceptual and mathematical understanding of the algorithms without the worry of other issues such as computational efficiency and accuracy when learnig the subject the first time. Through this process the students should gain not only the in-depth understanding of the theoretical background of the algorithms in terms of why and how they work, but also first-hand experiences of the implementation of these algorithms in terms of what type of problems an algorithm is designed to solve (or unable to solve), the pros and cons of a specific algorithm in comparion to other similar algorithms, and how to choose the proper values for the parameters of the algorithm for it to achieve the best performance. Inexperienced students' code may not be professionally developed with high computational efficiency and accuracy and good coding style, but once they have gained the insights of the algorithms, they can always take the full advantage of the professionally developed commercial software packages and apply them properly to solve real problems more effectively.

Moreover, they may potentially also be able to develop their own algorithms in the future to best solve specific problems they may have at hand.

For the students to test their own code implementing various algorithms discussed in the text, they need to apply their code to some datasets, such as those used for the examples in the book. Some of the datasets can actually be generated by the code provided in the homework, while a few others are provided on a CUP website for the book at.... The size and scale of such datasets may be too small to be practical, but they serve the purpose of debugging the code while avoiding long excution time if the code is applied to a large scale dataset. Once a piece of code is thoroughly debugged and tested, it can be applied to more realistic datasets of much larger scale, such as many of those publically available on the Internet for testing and benchmarking ML algorithms. The readers are highly encouraged to do so.

Structure and Organization of the Book

The book is organized in seven parts as listed below, although not all of them need to or can be covered in a typical one-semester course. Depending on the students' background, the chapters in Part I can be covered in the first few lectures dedicated to the background mathematics for the later chapters if the students are not already familiar with the topics, or, alternatively, these topics may only need to be quickly reviewed whenever they are needed while discussing the variou ML algorithms if the students are knowledgeable enough in these subjects.

Part II for regression needs to be taught early on before other topics, not only because the mathematics is relatively more straight forward than that in the later chapters, but also some basic issues discussed in this part (such as bias-variance tradeoff and regularization in Sections 4.2 and 4.3) are relevant in general to most learning methods covered in later chapters.

Part III for feature extraction can be treated as the preprocessing stage for the main learning algorithms to be discussed in Parts IV and V. Finally, Parts VI and VII as two independent and a little more advanced units can be covered in the later part of the course after the previous chapters are covered as their background.

- Part I *Mathematical Foundation*:

  This part is mostly dedicated to optimization, covering constrained as well as unconstrained methods, both of essential importance in ML, as many ML algorithms are formulated as an optimization problem to either minimize the difference between the output of a ML model and some observed training data, or to maximize the likelihood of the model parameters given the training data. Most ML textbooks do not discuss how such optimization methods are actually carried out, assuming the readers are already familiar with the subject or will pick up such knowledge on their own. However, optimization is a big subject of its own, a reader unfamiliar with it may not even know where to start. For this reason, the necessary

contents in optimization relevant to ML algorithms are covered in Part I of the book for the convenience of the reader.

Also included in Part I is the topic of solving equation systems, which is one of the most fundamental computational operations for many algorithms, and it is also most closely related to optimization, in the simple sense that a variable x that maximizes function $f(x)$ can be found by solving the equation $f'(x) = 0$, by, for example, the most widely used Newton's method.

- Part II *Regression Methods*:

  This part covers the most important regression algorithms widely used in ML, including learning and nonlinear regression, logistic and softmax regression, and Gaussian process regression. All such regression methods are closely related to classification, the core problem in ML, in the most straightforward sense that by simply thresholding the regression function, its domain is divided into two regions corresponding to two classes, i.e., the regression method can be readily used as a binary classifier. Moreover, a more sophisticated classifier can be obtained if the regression function is converted into the probability for a data sample to belong to one of the classes. This way we can not only classify a data sample into a class but also get the confidence or certainty of doing so.
- Part III *Feature Selection*:

  This part is about feature selection and dimensionality reduction, which can be considered as the preprocessing stage of the main ML operations of classification or clustering. The core method of this part is principal component analysis (PCA), by which the dimensionality of the dataset can be reduced significantly while the information pertaining to classification or clustering contained in the data is mostly conserved. We also consider a set of variations of the PCA methods, such as kernel PCA and probabilistic PCA. In this part we also consider the method of independent component analysis (ICA), by which the independent signal components can be restored from the mixture of their linear combinations.
- Part IV *Classification*:

  This part covers a set of core classification methods widely used in ML, including various supervised methods based on the training dataset containing a set of data samples each labeled by its class identity, such as the K-nearest neighbors (KNN) method and the minimum distance classifier, the adaptive boosting (AdaBoost), the naive Bayesian classifier, the support vector machine (SVM) with kernel mapping, and Gaussian process classifier, both binary and multiclass, and the decision tree method. In this part we also discuss unsupervised clustering methods without any training data, such as K-means method, and methods based on Gaussian and Bernoulli mixture models. All such classification and clustering can be considered as the partitioning of the multidimensional feature space into multiple regions each corresponding to one of the classes or clusters.

- Part V *Neural Networks*

  In Part V, we introduce a set of important algorithms based on artificial neural networks inspired by the biological neural network in the brain, including the Hebbian learning for association between two sets of patterns, the Hopfield network for self-association, the perceptron network based on kernel mapping for both binary and multiclass classification, and the most popular back propagation method for multiclass classification. In this part we also briefly discuss the idea of deep learning based on multilayer back propagation learning, typified by the convolutional neural network (CNN) for image object recognition. In addition, we also discuss unsupervised neural network algorithms such as competitive learning and the closely related self-organizing maps (SOM).

  We node that although the neural network methods covered in this part may seem conceptually very different from those regression and classification covered in previous parts, they may all look similar in the sense that the mathematical model of the learning taking place in a single layer of neural network is actually the same as a linear regression followed by a nonlinear mapping function, called activation function in the context of neural networks, such as a logistic mapping considered in Part II.

- Part VI *Reinforcement Learning*:

  This lart part of the book provides an introduction to the reinforcement learning (RL), an big area of active and ongoing study that has attracted a lot of interests in both research and applications. RL is considered as the third type of learning methods fundamentally diffenent from the two traditional of learning methods, the supervised learning (regression and classification) and unsupervised learning (clustering). Instead of assigning data points in the feature space into different classes or clusters as in supervised or unsupervised learning, RL aims at optimizing a sequence of actions for an agent to take for the purpose of receiving maximum returns from a given environment, based on its interaction with the environment. In this sense, RL can also be considered as a different type of optimization achieved by unsupervised interactive learning (without labeled training data). When combined with deep learning, RL has achieved amazing results in the recent years, most famously, the software AlphaGo that beat the best human player of the board game Go.

- Part VII *Large Language Models*:

  This part covers the latest advancements in large language models developed in the last few years. All of these models are based on the fundamental Transformer architecture first introduced in 2017, which, as a major breakthrough, revolutionized the field of natural language processing. Central to this transformation is the principal idea that "attention is all you need", allowing models to dynamically focus on relevant parts of the input data. This chapter delves into the intricacies of LLMs, exploring the underlying principles of backpropagation learning that enable these models to optimize

their performance effectively. We will examine various important concepts, such as self and cross-attention, transfer learning, and explore the issues of emergent properties and machine understanding that transcend simple correlations. We will also briefly introduce several important generative neural network models developed in recent years, beyond the Transformer models, for the general task of generating novel data samples based on patterns learned from existing datasets. These models have expanded the capabilities of machine learning to create novel and realistic synthetic data, including images, text, audio, and video, and have found applications across a wide range of fields.

Part I

Mathematical Fundations

Many machine learning problems can be formulated as an optimization problem to build a model of a set of observed data, called *training dataset* or simply *training set*, so that the model output matches the data in an optimal manner in certain sense. Typically this is done by either minimizing or maximizing an *objective function* of a set of model parameters, which measures the goodness of the model. For example, in either regression or classification, the goal of a certain type of learning algorithms is to minimize the error of the model, the difference between the model prediction and the given observed data, measured in a certain way, such as the squared error. In some other type of algorithms, the goal is to maximize the *likelihood* of the model parameters, defined as the probability of observing the given data conditioned on these parameters.

In the typical *supervised learning* paradigm, the modeling process can be formulated as to construct a function $y = f(\mathbf{x}, \boldsymbol{\theta})$ of some d variables as the components of a vector $\mathbf{x} = [x_1, \cdots, x_d]^T$, parameterized by a set of M parameters as the components of a vector $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_M]^T$. The goal is for this function to best fit a set of N data points in the training set, denoted by $\mathcal{D} = \{(\mathbf{x}_n, y_n), \ \ n = 1, \cdots, N\}$. While the form of the model function is usually assumed to be known based on some prior knowledge, such as a polynomial (e.g., linear or quadratic) function, or certain *probability density distribution (pdf)* of the data, the values of the M parameters in $\boldsymbol{\theta}$ are unknown and to be estimated based on the training set $\mathcal{D}$. The estimation can be carried out by various methods such as the following two that are widely used in machine learning:

- *Least squares estimation (LSE)*:
  The sum of squared error (SSE) between the model output and the desired output according to the training set is to be minimized:
  $$J_{sse}(\boldsymbol{\theta}) = \sum_{n=1}^{N} [y_n - f((\mathbf{x}_n, \boldsymbol{\theta}))]^2 \tag{0.1}$$
- *Maximum likelihood estimate (MLE)*:
  The likelihood of the parameter $\boldsymbol{\theta}$, the conditional probability of observing the data in $\mathcal{D}$ given a specific parameter $\boldsymbol{\theta}$, is to be maximized:
  $$J_{mle}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) \tag{0.2}$$

Solving either the minimization or maximization problem we get the optimal model parameter $\boldsymbol{\theta}^*$ for the model to best fit the observed data.

To prepare for the discussions of various machine learning algorithms in the future parts, such as how to find specifically the optimal $\boldsymbol{\theta}^*$ that minimizes/maximizes function $J(\boldsymbol{\theta})$ above, we will first consider in this part a set of typical methods for the optimization problem in the most generic form, find the optimal solution $\mathbf{x}^* = [x_1^2, \cdots, x_N^*]^T$ that either maximize or minimize a real valued multivariable function $y = f(\mathbf{x}) = f(x_1, \cdots, x_N)$. As maximizing $f(\mathbf{x})$ is equivalent to minimizing $-f(\mathbf{x})$, we typically only need to consider the minimization problem to

find the optimal solution $\mathbf{x}^*$ that minimizes $f(\mathbf{x})$, denoted by

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x}), \qquad \text{i.e.,} \qquad f(\mathbf{x}^*) = \min_{\mathbf{x}} f(\mathbf{x}) \leq f(\mathbf{x}) \tag{0.3}$$

Here the function $f(\mathbf{x})$ can be either linear or nonlinear, and an optimization problem is either *unconstrained* if the independent variable $\mathbf{x}$ of the function is allowed to take any value in the function domain $\mathbb{R}^N$, or *constrained* if $\mathbf{x}$ must be inside a subset of $\mathbb{R}^N$, called the *feasible region*. In machine learning, both linear and nonlinear methods, either constrained or not, are widely used. For example, the linear least squares method (LLS) for regression (to minimize the squared error such as that given in Eq. (0.1), Section 5.1) is solved as a linear unconstrained optimization problem, while the method of support vector machine (SVM) for classification (to maximize the margin between two classes, Section 14.1) is solved as a nonlinear constrained optimization problem.

In the following, we will review some necessary mathematical topics as the foundation for many of the learning methods to be discussed in future parts, including the numerical methods for solving equations in Chapter 1, which are not only important in their own right, but also most closely related to both unconstrained and constrained optimization problems in Chapters 2 and 3, which in turn will be needed for many important machine learning algorithms to be discussed in the future chapters.

1 Solving Equations

In this chapter, we consider some basic methods for solving a equation system in the most general form of M simultaneous equations of N variables:

$$\begin{cases} f_1(x_1, \cdots, x_N) = f_1(\mathbf{x}) = 0 \\ \cdots\cdots\cdots \\ f_M(x_1, \cdots, x_N)) = f_M(\mathbf{x}) = 0 \end{cases} \tag{1.1}$$

Here $\mathbf{x} = [x_1, \cdots, x_N]^T \in \mathbb{R}^N$ is an N-D column vector representing a point in the N-D vector space $\mathbb{R}^N$ spanned by $\{x_1, \cdots, x_N\}$, and $yf_m(\mathbf{x})$ $(m = 1, \cdots, M)$ is a scalar valued multivariate function that maps any point in its domain $\mathbf{x} \in \mathbb{R}^N$ to a scalar value y. This equation system can be represented more concisely in vector form

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{1.2}$$

where $\mathbf{f} = [f_1, \cdots, f_M]^T$ is a column vector containing the M functions as components and $\mathbf{0} = [0, \cdots, 0]^T$ is a vector containing M zeros. Solving this equation system means to find a solution $\mathbf{x}^* \in \mathbb{R}^N$ that satisfies all M equations. If such a solution does not exist, we would still like to find an approximate solution by which certain error, e.g., the sum of squared error (SSE) $||\mathbf{f}(\mathbf{x})||^2$, is minimized.

Geometrically, a function $y = f(\mathbf{x})$ defined over domain $\mathbb{R}^N$ is a *hypersurface* in an N+1 dimensional space, of which the N+1st dimension represents the function value $y = f(\mathbf{x})$ corresponding to a point $\mathbf{x} \in \mathbb{R}^N$. Specially the hypersurface is a curve if $N = 1$ or a surface if $N = 2$. The *roots* or *zeros* of function $f(\mathbf{x})$, i.e., the solutions of equation $f(\mathbf{x}) = 0$, are composed of all points on the the intersection, if exists, of the hypersurface $y = f(\mathbf{x})$ and the hyperplane $y = 0$.

For example, in the special case of $M = N = 2$, functions $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ are two surfaces defined over the 2-D space spanned by x_1 and x_2, and the roots of each of the two functions are on a curve as the intersection of the corresponding surface and the 2-D space. The solutions of the system composed of the two equations $f_1 = 0$ and $f_2 = 0$ are the intersection of these two curves in the 2-D space if they do intersect, otherwise no solution exists.

Example 1.1 Consider a simultaneous equation system with $M = N = 2$:

$$\begin{cases} f_1(x_1, x_2) = x_1^2 + x_2^2 - 2 = 0 \\ f_2(x_1, x_2) = x_1 + x_2 - C = 0 \end{cases}$$

The first function $f_1(x_1, x_2)$ is a parabolic cone in the $N+1=3$ dimensional space centrally symmetric to the vertical axis, and its roots form a circle $x_1^2+x_2^2=2$ on the 2-D plane spanned by x_1 and x_2 centered at the origin (0, 0) with radius 1; the second function $f_2(x_1, x_2)$ is a plane in the 3-D space through the origin, and its roots form a straight line $x_2 = C - x_1$ on the 2-D plane. The solutions of the equation system are where the two curves intersect:

- If $C = 0$, there are two solutions (1, −1) and (−1, 1);
- If $C = 2$, there is only one solution (1, 1);
- If $C = 3$, the two curves do not intersect, i.e., no solution exists.

1.1 Linear Equation Systems

Consider a linear equation system of M equations and N variables:

$$\begin{cases} f_1(\mathbf{x}) & = f_1(x_1, \cdots, x_N) & = \sum_{j=1}^{N} a_{1j}x_j - b_1 = 0 \\ \cdots & \cdots & \cdots \\ f_M(\mathbf{x}) & = f_1(x_1, \cdots, x_N) & = \sum_{j=1}^{N} a_{Mj}x_j - b_M = 0 \end{cases} \tag{1.3}$$

which can be expressed in vector form:

$$\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0} \tag{1.4}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \cdots & a_{MN} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_M \end{bmatrix} \tag{1.5}$$

In general, the existence and uniqueness of the solution can be determined based on the *fundamental theorem of linear algebra* (Table 3 in Section A.5), depending on the rank R of the coefficient matrix $\mathbf{A}$, and whether the system is underdetermined (underconstrained) if $M < N$, or overdetermined (overconstrained) if $M > N$.

If $\mathbf{A}$ is a full rank square matrix with $R = M = N$, then its inverse $\mathbf{A}^{-1}$ exists and the system has a unique solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. But if $M > N = R$ and $\mathbf{b}$ is not in the column space of $\mathbf{A}$, the system is overconstrained with no solutions. In this case we can still find an optimal approximate solution that minimizes the *sum-of-squares error (SSE)*, called *objective function* in general optimization problems, defined as

$$\begin{aligned} \varepsilon(\mathbf{x}) &= \frac{1}{2}||\mathbf{r}||^2 = \frac{1}{2}\mathbf{r}^T\mathbf{r} = \frac{1}{2}(\mathbf{b} - \mathbf{A}\mathbf{x})^T(\mathbf{b} - \mathbf{A}\mathbf{x}) \\ &= \frac{1}{2}\left(\mathbf{b}^T\mathbf{b} - \mathbf{x}^T\mathbf{A}^T\mathbf{b} - \mathbf{b}^T\mathbf{A}\mathbf{x} + \mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}\right) \end{aligned} \tag{1.6}$$

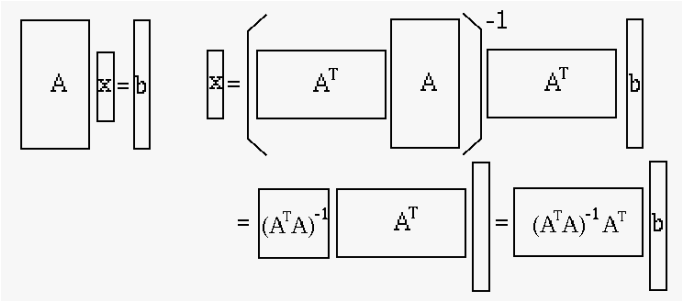


Figure 1.1 The Pseudo Inverse (Section 1.1)

where $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ is the *residual* of the system, and $||\mathbf{r}||^2 = \sum_{m=1}^{M} r_m^2$ (thereby the term sum of squares error). The optimal solution $\mathbf{x}^*$ that minimizes $\varepsilon(\mathbf{x})$ can be found by setting its derivative with respect to $\mathbf{x}$ to zero (see Section A.4.3 for differentiation with respect to a vector variable):

$$\begin{aligned}\frac{d}{d\mathbf{x}}\varepsilon(\mathbf{x}) &= \frac{1}{2}\frac{d}{d\mathbf{x}}||\mathbf{r}||^2 = \frac{1}{2}\frac{d}{d\mathbf{x}}\left(\mathbf{b}^T\mathbf{b} - \mathbf{x}^T\mathbf{A}^T\mathbf{b} - \mathbf{b}^T\mathbf{A}\mathbf{x} + \mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}\right) \\ &= -\mathbf{A}^T\mathbf{b} + \mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{0}\end{aligned} \tag{1.7}$$

Solving this matrix equation we get the optimal approximate solution

$$\mathbf{x}^* = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b} = \mathbf{A}^-\mathbf{b} \tag{1.8}$$

As illustrated in Fig. 1.1. Here $\mathbf{A}^- = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$ is the *pseudo-inverse* (see Section A.6.1) of the non-square matrix $\mathbf{A}$. Here we assume the M equations are independent, i.e., the $N \times N$ square matrix $\mathbf{A}^T\mathbf{A}$ has a full rank $R = N$, i.e., it is nonsingular and invertible with all eigenvalues being non-zero.

If the M equations are barely independent, i.e., some eigenvalues of $\mathbf{A}^T\mathbf{A}$ are very close to zero, then it is near-singular but still invertible, and some eigenvalues of its inverse $(\mathbf{A}^T\mathbf{A})^{-1}$ may take huge values. Consequently the result in Eq. (1.8) may vary greatly due to some small random fluctuation in either $\mathbf{A}$ or $\mathbf{b}$. In this case, the system is said to be *ill-conditioned* or *ill-posed*, as it is prone to noise, i.e., its solution is highly sensitive to noise and therefore unreliable and unstable.

Such an ill-posed problem can be addressed by *regularization*, so that the solution $\mathbf{x}$ is forced to avoid taking huge values and therefore less sensitive to noise. This can be done by including in the objective function an additional penalty term for large values of $\mathbf{x}$ scaled by a parameter λ:

$$J(\mathbf{x}) = \frac{1}{2}||\mathbf{r}||^2 + \frac{\lambda}{2}||\mathbf{x}||^2 \tag{1.9}$$

By minimizing $J(\mathbf{x})$, we can obtain a solution $\mathbf{x}$ of small norm $||\mathbf{x}||$ as well as a small error $\varepsilon(\mathbf{x})$. Just as before, the solution $\mathbf{x}$ can be obtained by setting the

derivative of $J(\mathbf{x})$ to zero

$$\frac{d}{d\mathbf{x}}J(\mathbf{x}) = -\mathbf{A}^T\mathbf{b} + \mathbf{A}^T\mathbf{A}\mathbf{x} + \lambda\mathbf{x} = \mathbf{0} \tag{1.10}$$

and solving the resulting equation to get:

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^T\mathbf{b} \tag{1.11}$$

We see that even if $\mathbf{A}^T\mathbf{A}$ is near singular, matrix $\mathbf{A}^T\mathbf{A} + \lambda\mathbf{I}$ is not due to the additional term $\lambda\mathbf{I}$.

Here λ is called the *hyperparameter*, by adjusting it we can make a proper tradeoff between accuracy and stability:

- Small λ: the solution is more accurate but less stable as it is more prone to noise, i.e., the variance error may be large. This is called *overfitting*;
- Large λ: the solution is less accurate but more stable as it is less affected by noise. This is called *underfitting*.

The issue of overfitting versus underfitting is of essential importance in machine learning in general, it will appear repeatedly and be specifically addressed while discussing various regression and classification algorithms in the later chapters.

1.2 The Bisection and Secant Methods

Most of equation systems of interest are nonlinear. Unlike the linear equation systems which can be solved under the guidance of the fundamental theorem of linear algebra, for nonlinear equation systems, there exists neither a theory regarding the existence and uniqueness of their solutions, nor closed-form solutions in general. We therefore have to rely on numerical methods to find one or more solutions, if they do exist, in an iterative manner from certain initial guess of the solution, if the iteration converges, i.e., the difference between two consecutive results eventually approach to zero.

We first consider some methods that find a solution of a single-variable nonlinear equation $f(x) = 0$, by searching iteratively through a neighborhood of the domain, in which a solution is known to be located.

1.2.1 The Bisection Search

This method requires two initial guesses $x_0 < x_1$ satisfying $f(x_0)f(x_1) < 0$. As $f(x_0)$ and $f(x_1)$ are on opposite sides of the x-axis $y = 0$, the solution x^* at which $f(x^*) = 0$ must reside somewhere in between of these two guesses, i.e., $x_0 < x^* < x_1$.

Given such two end points x_0 and x_1, we could find any point x_2 in between and use it to replace one of the end points at which the function value $f(x)$ has the same sign as that of $f(x_2)$. The new search interval is $a = x_2 - x_0$ if

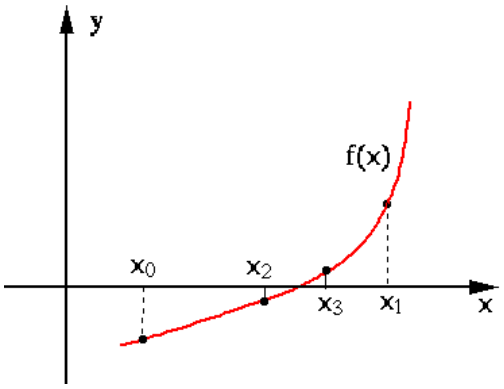


Figure 1.2 Bisection Search

$f(x_2)f(x_1) > 0$, or $b = x_1 - x_2$ if $f(x_0)f(x_2) > 0$, either of which is smaller than the previous interval $a + b = x_1 - x_0$. This process is then carried out iteratively until the solution is eventually approached, with a guaranteed convergence.

To avoid the worst possible case in which the solution always happens to be in the larger of the two sections a and b, we typically choose x_2 to be the middle point $x_2 = (x_0 + x_1)/2$ between the two end points, so that $a = b = (x_1 - x_0)/2$, i.e., the new search interval is always halved in each step of the iteration:

$$x_{n+1} = \frac{x_n + x_{n-1}}{2} \tag{1.12}$$

as illustrated in Fig. 1.2. Here, without loss of generality, we have assumed $f(x_{n-1})f(x_{n+1}) > 0$ and x_{n-1} is replaced by x_{n+1}.

For example, we assume the root is located at $x^* = 2.2$ between the two initial values are $x_0 = 0$ and $x_1 = 8$, then we get

n	0	1	2	3	4	5	6	7	8
x_n	0	8	4	2	3	2.5	2.25	2.125	$\cdots$
$\lvert e_n\rvert$	2.2	5.8	1.8	0.2	0.8	0.3	0.05	0.075	$\cdots$

(1.13)

Note that the error $|e_n| = |x_n - x^*|$ does not necessarily always reduce monotonically. However, as in each iteration the search interval is always halved, the worst possible error is also halved:

$$|e_{n+1}| = |x_{n+1} - x^*| \le \frac{|x_n - x_{n-1}|}{2} = \frac{|x_0 - x_1|}{2^n} \approx \frac{|e_n|}{2} \tag{1.14}$$

In general, to measure how quickly or slowly an iterative method converges, we consider the following limit of the ratio of errors of two consecutive iterations:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^q} = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|} \le \mu \tag{1.15}$$

where q is the *order of convergence* and μ is the *rate of convergence*. The higher

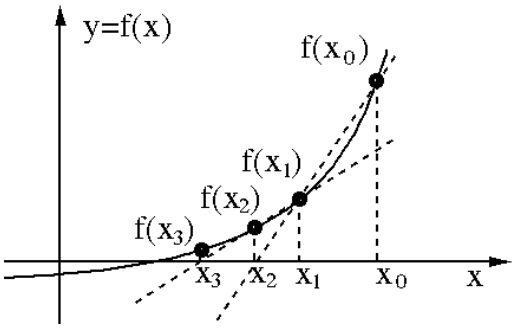


Figure 1.3 The Secant Method

the order q, the more rapidly the iteration converges. For the bisection method, we have

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^q} = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|} \le \mu = \frac{1}{2} \tag{1.16}$$

We see that its order of convergence is $q = 1$ (it converges linearly) with the rate of convergence $\mu = 1/2$. Compared to other methods to be considered later, the linear convergence of the bisection method is rather slow, but its has the advantage that no derivative $f'(x)$ of the given function is needed and the given function does not need to be differentiable.

1.2.2 The Secant Method

Same as in the bisection method, we assume there are two initial values x_0 and x_1 available, but they no longer have to satisfy $f(x_0)f(x_1) < 0$. Here the iteration is based on the zero-crossing of the secant line passing through the two points $f(x_0)$ and $f(x_1)$, instead of the middle point between them. The equation for the secant is:

$$y = f(x_0) + \frac{x - x_0}{x_1 - x_0}(f(x_1) - f(x_0)) = f(x_0) + \hat{f}'(x_0)(x - x_0) \tag{1.17}$$

where $\hat{f}'(x_0)$ is the finite difference that approximates the derivative $f'(x_0)$:

$$\hat{f}'(x_0) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{\Delta f(x_0)}{\Delta x_0} \quad \stackrel{\Delta x_0 \to 0}{\Longrightarrow} \quad f'(x_0) \tag{1.18}$$

Setting $y = 0$ in the equation above, we get the zero crossing point:

$$x = x_0 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_0) = x_0 - \frac{f(x_0)}{\hat{f}'(x_0)} \tag{1.19}$$

As shown in Fig. 1.3, the new value x obtained above is a better estimate of the root than either x_0 or x_1, and is used to replace one of them:

- If $f(x_0)f(x_1) < 0$, then x_i $(i = 0, 1)$ satisfying $f(x_i)f(x) > 0$ is replaced by $x_2 = x$ (same as the bisection method);
- If $f(x_0)f(x_1) > 0$, then x_i $(i = 0, 1)$ with greater $|f(x_i)|$ is replaced by $x_2 = x$.

This process is then carried out iteratively to approach the root:

$$x_{n+1} = x_n - \frac{f(x_n)}{\hat{f}'(x_n)} \tag{1.20}$$

If the derivative $f'(x)$ is available, it can be used to replace its approximation $\hat{f}'(x_n)$ and the iteration becomes the Newton-Raphson method (Section 1.4):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.21}$$

In case $f(x_n) = f(x_{n-1})$, the approximated derivative is zero $\hat{f}'(x_n) = 0$, and x_{n+1} cannot be found. In such a case, a root may exist between x_n and x_{n-1}. Therefore the way to resolve this problem is to combine the bisection search with the secant method so that when $\hat{f}' = 0$, the algorithm will switch to bisection search. This is Dekker's method:

$$x_{n+1} = \begin{cases} x_n - f(x_n)/\hat{f}'(x_n) & \text{if } f(x_n) \neq f(x_{n-1}) \\ (x_n + x_{n-1})/2 & \text{otherwise} \end{cases} \tag{1.22}$$

The order of convergence of the secant method is $q = 1.618$ as shown in Appendix 1 of this chapter, i.e., the secant method is converges more quickly than the bisection search method with $q = 1$ (which does not take advantage of the information of the specific function $f(x)$).

1.3 Fixed-point Iteration

Fixed-point iteration is a method of finding the fixed point of a given function. It can be used to solve the generic equation system $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, as seen in this section. In particular, it will be used to solve the Bellman equation in dynamic programming, as we will see in Chapter 20.

We first consider the special case of solving a single-variable nonlinear equation $f(x) = 0$ $(M = N = 1)$. To find a solution x^* that satisfies the equation, we could first convert it into an equivalent equation $g(x) = x$, so that an x satisfying one of the equations will also satisfy the other. We then carry out an iteration $x_{n+1} = g(x_n)$ from some initial value x_0. If the iteration converges at a point x^*, i.e., $g(x^*) = x^*$, then we also have $f(x^*) = 0$, i.e., x^* is a solution of the equation $f(x) = 0$. Consider the following two examples:

Example 1.2

$$f(x) = \log(x) - 0.5 = 0$$

To solve the equation, we construct another function

$$g_1(x) = x - f(x) = x - \log(x) + 0.5$$

so that the equation $g(x) = x$ is equivalent to the given equation $f(x) = 0$ (left Fig. 1.4). We then carry out the iteration $x_{n+1} = g(x_n)$ from an initial value, such as $x_0 = 0.5$, and get

n	x_n	$g(x_n)$	$f(x_n)$
0	0.50000	1.69315	−1.19315
1	1.69315	1.66656	0.02659
2	1.66656	1.65580	0.01076
3	1.65580	1.65152	0.00428
4	1.65152	1.64982	0.00169
5	1.64982	1.64915	0.00067
6	1.64916	1.64889	0.00026
7	1.64889	1.64879	0.00010
8	1.64879	1.64875	0.00004
9	1.64875	1.64873	0.00002
10	1.64873	1.64873	0.00001
11	1.64873	1.64872	0.00000
12	1.64872	1.64872	0.00000

We see that the iteration converges to $x^* = \lim_{n\to\infty} g(x_n) = 1.64872$ satisfying $g(x^*) = x^* - f(x^*) = x^*$, and, equivalently, x^* is also the solution of the given equation $f(x) = 0.5 - \log(x) = 0$:

$$0.5 - log(1.64872) = 0, \qquad \text{i.e.,} \qquad e^{0.5} = \sqrt{e} = 1.64872$$

Alternatively, we could construct a different function

$$g'(x) = x + f(x) = x + \log(x) - 0.5$$

also equivalent to $f(x) = 0$ (right Fig. 1.4). But this iteration no longer converges!

Example 1.3

$$f(x) = e^x - 1/x = 0$$

Take logarithm of the equation, we get an equivalent form (left in Fig. 1.5)

$$g(x) = x = e^{-x}$$

and an iteration:

$$x_{n+1} = g(x_n) = e^{-x_n} \overset{n\to\infty}{\Longrightarrow} x^* = 0.5671$$

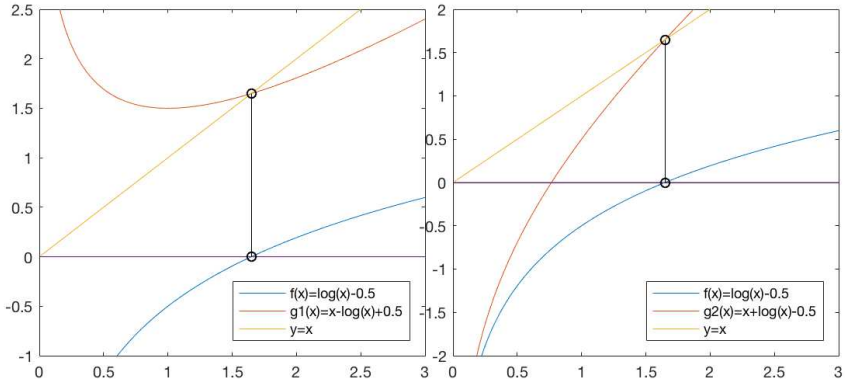


Figure 1.4 Example 1.2: Iteration converges for $g(x)$ (left) but diverges for $g'(x)$ (right)

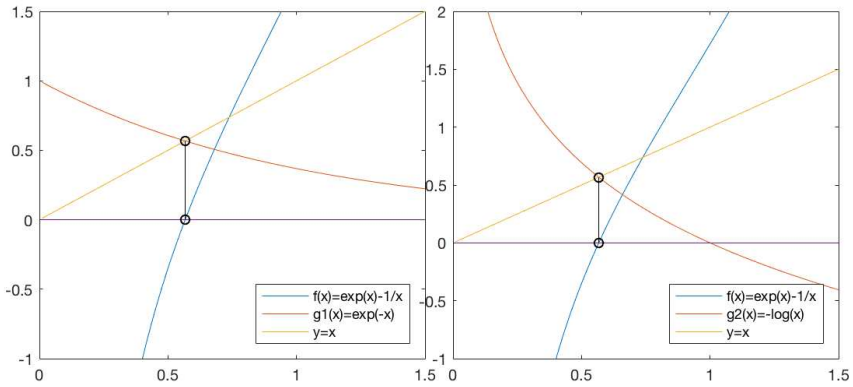


Figure 1.5 Example 1.3: Iteration converges for $g(x)$ (left) but diverges for $g'(x)$ (right)

that converges to the root of the given equation $f(x) = e^x - 1/x = 0$, i.e., $e^{-0.5671} = 0.5671$.

However, if the given equation is convert into an alternative form $g'(x) = x = -\ln(x)$ (right in Fig. 1.5), the iteration no longer converges.

Why does this iterative method work in some cases but fail in other? To answer these questions we need to understand the theory behind the method, the *fixed point* of a *contraction function*.

If a single variable function $g(x)$ satisfies

$$|g(x_1) - g(x_2)| \leq k|x_1 - x_2|, \qquad k \geq 0 \tag{1.23}$$

it is said to be *Lipschitz continuous*, and k is a *Lipschitz constant*. If $k = 1$, then $g(x)$ is a *non-expansive* function, if $0 \leq k < 1$, then $g(x)$ is a *contraction function* or simply a *contraction*. These concepts can be generalized to solving multivariate equations $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

Definition: In a metric space V with certain distance $d(\mathbf{x}, \mathbf{y})$ defined between any two points $\mathbf{x}, \mathbf{y} \in V$, a function $\mathbf{g} : V \to V$ is a *contraction* if

$$d(\mathbf{g}(\mathbf{x}), \mathbf{g}(\mathbf{y})) \leq k\, d(\mathbf{x}, \mathbf{y}) \tag{1.24}$$

The smallest k value that satisfies the above is called the *Lipschitz constant*.

Intuitively, a contraction reduces the distance between points in the space, i.e., it brings them closer together. A function $\mathbf{g}(\mathbf{x})$ may not be a contraction through out its entire domain, but it can be a contraction in the neighborhood of a certain point $\mathbf{x}^* \in V$, composed of all points $\mathbf{x}$ satisfying $d(\mathbf{x}, \mathbf{x}^*) < \epsilon > 0$ (i.e., $\mathbf{x}$ is a neighbor of $\mathbf{x}^*$ as defined by ϵ) so that

$$d(\mathbf{g}(\mathbf{x}), \mathbf{g}(\mathbf{x}^*)) \leq k\, d(\mathbf{x}, \mathbf{x}^*), \qquad 0 \leq k < 1 \tag{1.25}$$

Definition: A *fixed point* $\mathbf{x}^*$ of a function $\mathbf{g}(\mathbf{x})$ is a point in its domain that is mapped to itself:

$$\mathbf{x}^* = \mathbf{g}(\mathbf{x}^*) \tag{1.26}$$

We immediately have

$$\mathbf{g}(\mathbf{g}(\mathbf{x}^*)) = \mathbf{g}^2(\mathbf{x}^*) = \mathbf{x}^*, \cdots\cdots, \mathbf{g}(\mathbf{g}(\mathbf{g}(\cdots \mathbf{g}(\mathbf{x}^*) \cdots))) = \mathbf{g}^n(\mathbf{x}^*) = \mathbf{x}^* \tag{1.27}$$

A fixed point $\mathbf{x}^*$ is an *attractive fixed point* if any point $\mathbf{x}$ in its neighborhood converges to $\mathbf{x}^*$, i.e., $\lim_{n\to\infty} \mathbf{g}^n(\mathbf{x}) = \mathbf{x}^*$.

We further consider two theorems regarding the fixed point and a contraction mapping.

Fixed-Point Theorem : Let $\mathbf{g}(\mathbf{x})$ be a contraction satisfying

$$d(\mathbf{g}(\mathbf{x}) - \mathbf{g}(\mathbf{y})) \leq k\, d(\mathbf{x} - \mathbf{y}) \qquad (0 \leq k < 1) \tag{1.28}$$

then there exists a unique fixed point $\mathbf{x}^* = \mathbf{g}(\mathbf{x}^*)$, which can be found by an iteration from an arbitrary initial point $\mathbf{x}_0$:

$$\lim_{n\to\infty} \mathbf{x}_n = \lim_{n\to\infty} \mathbf{g}(\mathbf{x}_{n-1}) = \lim_{n\to\infty} \mathbf{g}^2(\mathbf{x}_{n-2}) = \cdots = \lim_{n\to\infty} \mathbf{g}^n(\mathbf{x}_0) = \mathbf{x}^* \tag{1.29}$$

Contraction Mapping Theorem: Let $\mathbf{x}^* = \mathbf{g}(\mathbf{x}^*)$ be a fixed point of a differentiable function $\mathbf{g}(\mathbf{x})$, i.e, $\partial g_i/\partial x_j$ exists for any $1 \leq i, j \leq N$. If the norm of the Jacobian matrix is smaller than 1, $||\mathbf{J}_\mathbf{g}(\mathbf{x}^*)|| < 1$, then $\mathbf{g}(\mathbf{x})$ is a contraction at $\mathbf{x}^*$.

The Jacobian matrix of $\mathbf{g}(\mathbf{x})$ is defined as

$$\mathbf{g}'(\mathbf{x}) = \mathbf{J}_\mathbf{g}(\mathbf{x}) = \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_N}{\partial x_1} & \cdots & \frac{\partial g_N}{\partial x_N} \end{bmatrix} \tag{1.30}$$

The proofs of the two theorems above are given in Appendix 3 of the chapter.

In the special case of a single-variable function $g(x)$ with $N = 1$, its Jacobian is simply its derivative $g'(x)$, and we have

$$g(x) - g(x^*) = g'(x^*)(x - x^*) + \frac{1}{2} g''(x^*)(x - x^*)^2 + R(x - x^*) \tag{1.31}$$

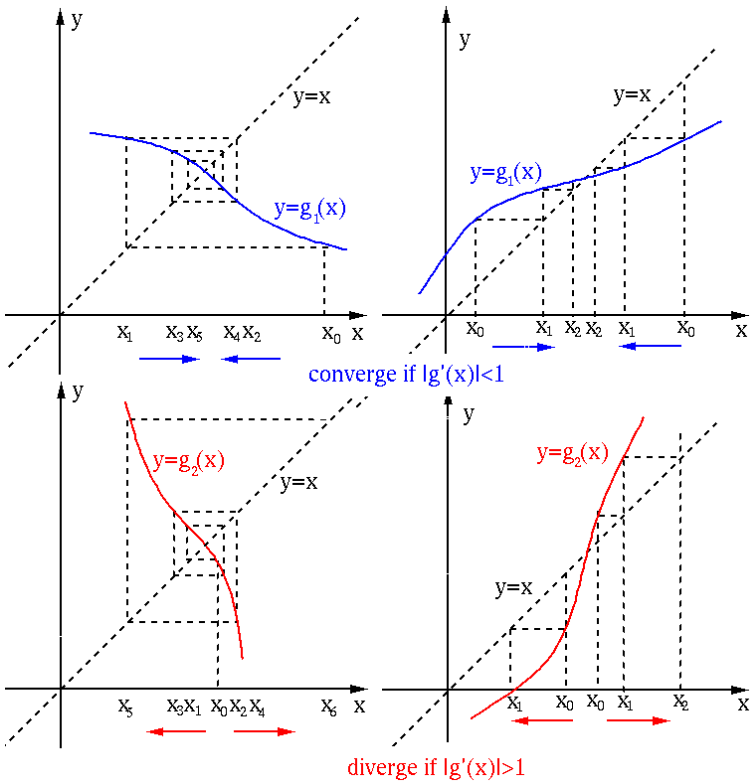


Figure 1.6 Convergence (top) and Divergence (bottom) of the Fixed Point Method

and

$$|g(x) - g(x^*)| \leq |g'(x^*)(x - x^*)| \leq |g'(x^*)|\,|(x - x^*)| \tag{1.32}$$

If $|g'(x)| < 1$, then $g(x)$ is a contraction at x^*.

Now we should understand why in the examples of the previous section the iteration leads to convergence in some cases but divergence in other cases: if $|g'(x)| < 1$, the iteration will converge to the root x^* of $f(x) = 0$, but if $|g'(x)| > 1$, it never will never converge.

The iterative process $x_{n+1} = g(x_n)$ for finding the fixed of a single-variable function $g(x)$ is shown in Fig. 1.6 as the intersections of the function $y = g(x)$ and the identity function $y = x$. If $|g'(x)| < 1$, the iteration converges as shown in the two cases on top, while if $|g'(x)| > 1$ it diverges as shown in the two cases at the bottom.

In general the fixed-point iteration converges linearly:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|} = \lim_{n\to\infty} \frac{|x_{n+1} - x^*|}{|x_n - x^*|} = \lim_{n\to\infty} \frac{|g(x_n) - g(x^*)|}{|x_n - x^*|} = |g'(x^*)| = \mu < 1 \tag{1.33}$$

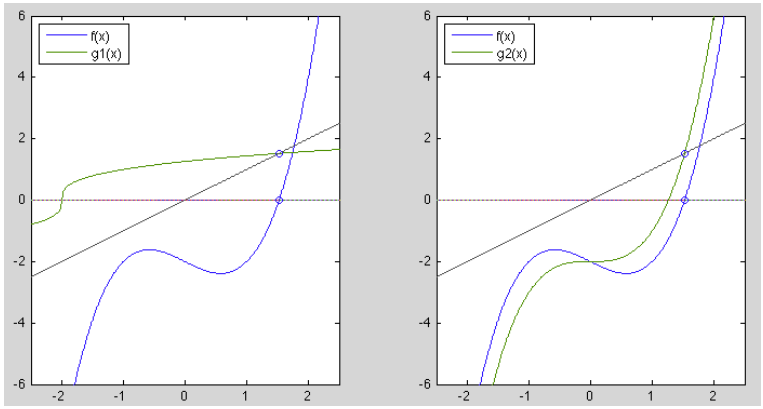


Figure 1.7 Example 1.4, plots of $g_1(x)$ (left) and $g_2(x)$ (right)

However, if the iteration function $g(x)$ has zero derivative at the fixed point $g'(x^*) = 0$, we have

$$\begin{aligned} g(x) - g(x^*) &= g'(x^*)(x - x^*) + \frac{1}{2}g''(x^*)(x - x^*)^2 + R(x - x^*) \\ &= \frac{1}{2}g''(x^*)(x - x^*)^2 + R(x - x^*) \end{aligned} \tag{1.34}$$

and the iteration converges quadratically:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^2} = \lim_{n\to\infty} \frac{|x_{n+1} - x^*|}{|x_n - x^*|^2} = \lim_{n\to\infty} \frac{|g(x_n) - g(x^*)|}{|x_n - x^*|^2} = \frac{1}{2}|g''(x^*)| = \text{constant} \tag{1.35}$$

Moreover, if $g''(x^*) = 0$, then the iteration converges cubically.

Example 1.4

$$f(x) = x^3 - x - 2 = 0$$

This equation can converted into an equivalent form of $g(x) = x$ in two different ways:

$$g_1(x) = x = \sqrt[3]{x + 2}, \qquad \text{and} \qquad g_2(x) = x = x^3 - 2$$

These two functions are plotted in Fig. 1.7, together with $f(x)$ and the identity function. The iteration based on $g_1(x)$ (left) converges to the solution $x^* = 1.5214$ for any initial guess $-\infty < x_0 < \infty$, as $g_1(x)$ is a contraction. However, the iteration based on $g_2(x)$ (right) does not converge to x^* as it is not a contraction in the neighborhood of x^*. In fact, the iteration will diverge towards either $-\infty$ if $x_0 < x^*$ or ∞ if $x_0 > x^*$.

Example 1.5

$$f(x) = x^3 - 3x + 1 = 0$$

we first convert it into the form of $g(x) = x$ in two different ways:

$$g_1(x) = x = \sqrt[3]{3x-1}, \qquad g_2(x) = x = \frac{1}{3}(x^3 - 1)$$

As can be seen in Fig. 1.8, this equation has three solutions,

$$x_1 = -1.8794, \qquad x_2 = 0.3473, \qquad x_3 = 1.5321$$

of which x_1 and x_3 can be obtained by the iteration based on $g_1(x)$ and x_2 can be obtained by the iteration based on $g_2(x)$. But neither of them can find all three roots.

- As shown in the plot on the left, $|g_1'(x)| < 1$ for all $-\infty < x < \infty$ except in the neighborhood of $x_2 = 0.3473$, i.e., $g_1(x)$ is a contraction mapping everywhere except around x_2. Therefore the iteration starting from any initial guess x_0 will converge to either $x_1 = -1.8749$ if $x_0 < x_2$, or $x_3 = 1.5321$ if $x_0 > x_2$.

$$x_{n+1} = g_1(x_n) \overset{n\to\infty}{\Longrightarrow} \begin{cases} x_1 = -1.8749 & x_0 < x_2 \\ x_3 = 1.5321 & x_0 > x_2 \end{cases}$$

  However, as $g_1(x)$ is not a contraction mapping around $x = x_2$, the iteration will never converge to x_2.
- As shown in the plot on the right, $|g_2'(x)| > 1$ for all $-\infty < x < \infty$ except in the neighborhood of x_2, i.e., $g_2(x)$ is not a contraction mapping around either x_1 or x_3. Therefore the iteration based on $g_2(x)$ will not converge to either x_1 or x_3, but it may converge to x_2, if the initial guess x_0 is in the range $x_1 < x_0 < x_3$. However, if x_0 is outside this range the iteration will diverge toward either $-\infty$ if $x_0 < x_1$ of ∞ if $x_0 > x_3$.

$$x_{n+1} = g_2(x_n) \overset{n\to\infty}{\Longrightarrow} \begin{cases} -\infty & x_0 < x_1 \\ x_2 = 0.3473 & x_1 < x_0 < x_3 \\ \infty & x_0 > x_3 \end{cases}$$

n	x_n	$f(x_n)$
1	$-2.0408275e+00$	$-1.3775173e+00$
2	$-1.9240239e+00$	$-3.5041089e-01$
3	$-1.8919392e+00$	$-9.6254050e-02$
4	$-1.8829328e+00$	$-2.7019276e-02$
5	$-1.8803890e+00$	$-7.6311660e-03$
6	$-1.8796694e+00$	$-2.1590449e-03$
7	$-1.8794656e+00$	$-6.1114717e-04$
8	$-1.8794080e+00$	$-1.7301761e-04$
9	$-1.8793916e+00$	$-4.8983741e-05$
10	$-1.8793870e+00$	$-1.3868146e-05$
11	$-1.8793857e+00$	$-3.9263250e-06$
12	$-1.8793853e+00$	$-1.1116151e-06$

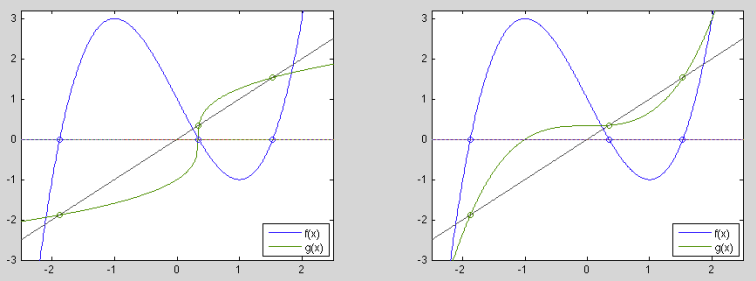


Figure 1.8 Example 1.5, plots of $g_1(x)$ (left) and $g_2(x)$ (right)

Example 1.6

$$f(x) = \sin(x) - x^3 = 0$$

This equation can be converted into the form $g(x) = x$ in different ways:

- $$g_0(x) = x = \sqrt[3]{\sin(x)}, \quad g_0'(x) = \frac{\cos(x)}{3\,\sin(x)^{2/3}}$$

n	x_n	$f(x_n)$
0	$2.000000e+00$	-7.0907
1	$9.688027e-01$	-0.085089
2	$9.375886e-01$	-0.018075
3	$9.306842e-01$	-0.0041048
4	$9.291018e-01$	-0.00094611
5	$9.287364e-01$	-0.00021881
6	$9.286518e-01$	$-5.0647e-05$
7	$9.286322e-01$	$-1.1725e-05$
8	$9.286277e-01$	$-2.7144e-06$

- $$g_1(x) = x = \sin(x)/x^2, \quad g_1'(x) = \frac{\cos(x)}{x^2} - \frac{2\,\sin(x)}{x^3}$$
- $$g_2(x) = x + \sin(x) - x^3, \quad g_2'(x) = \cos(x) - 3\,x^2 + 1$$
- $$g_3(x) = x - \frac{\sin(x) - x^3}{\cos(x) - 3x^2}, \quad g_3'(x) = -\frac{(6x + \sin(x))(\sin(x) - x^3)}{(\cos(x) - 3x^2)^2}$$

n	x_n	$f(x_n)$
0	$2.000000e+00$	-7.0907
1	$1.428913e+00$	-1.9276
2	$1.106787e+00$	-0.46152
3	$9.637850e-01$	-0.073886
4	$9.304466e-01$	-0.0036294
5	$9.286316e-01$	$-1.0509e-05$
6	$9.286263e-01$	$-8.9029e-11$

Example 1.7

$$f(x) = e^x - \frac{1}{x}$$

- $$g_0(x) = x = e^{-x}, \quad g_0'(x) = e^{-x}$$

  The iteration from any initial guess $x_0 > 0$ will converge to $x^* = g_1(x^*) = 0.56714$.

- $$g_1(x) = x = -\log(x), \quad g_1'(x) = 1/x$$

  Around $x = x^* = 0.56714$, $g_1'(x) = 1/x > 1$, the iteration does not converge.

- $$g_2(x) = x - \frac{e^x - 1/x}{e^x + 1/x^2}, \quad g_2'(x) = \frac{(e^x - 1/x)(e^x - 2/x^3)}{(e^x + 1/x^2)^2}$$

Example 1.8 Consider a 3-variable linear vector function $\mathbf{f}(\mathbf{x})$ of arguments $\mathbf{x} = [x,\ y,\ z]^T$:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \qquad \begin{cases} f_1(\mathbf{x}) = 6x + 3y + 2z - 18 = 0 \\ f_2(\mathbf{x}) = 2x + 7y + 3z - 25 = 0 \\ f_3(\mathbf{x}) = x + 3y + 5z - 22 = 0 \end{cases}$$

from which the g-function can be obtained:

$$\mathbf{g}(\mathbf{x}) = \mathbf{x}, \qquad \begin{cases} g_1(\mathbf{x}) = x = -(3y + 2z - 18)/6 \\ g_2(\mathbf{x}) = y = -(2x + 3z - 25)/7 \\ g_3(\mathbf{x}) = z = -(x + 3y - 22)/5 \end{cases}$$

The Jacobian $\mathbf{g}(\mathbf{x})$ of this linear system is a constant matrix

$$\mathbf{J}_g = \begin{bmatrix} 0 & -1/2 & -1/3 \\ -2/7 & 0 & -3/7 \\ -1/5 & -3/5 & 0 \end{bmatrix}$$

with the induced p=2 norm (maximum singular value) $||\mathbf{J}_g|| = 0.851 < 1$. Consequently, the iteration $\mathbf{x}_{n+1} = \mathbf{g}(\mathbf{x}_n)$ converges from an initial guess $\mathbf{x}_0 = [1,\ 1,\ 1]^T$ to the solution $\mathbf{x} = [1,\ 2,\ 3]^T$.

Alternatively, the g-function can also be obtained as

$$\mathbf{g}'(\mathbf{x}) = \mathbf{x}, \qquad \begin{cases} g_1'(\mathbf{x}) = x = -(3y + 5z - 22) \\ g_2'(\mathbf{x}) = y = -(2x + 3z - 25)/7 \\ g_3'(\mathbf{x}) = z = -(6x + 3y - 18)/2 \end{cases}$$

The Jacobian is

$$\mathbf{J}_{g'} = \begin{bmatrix} 0 & -3 & -5 \\ -2/7 & 0 & -3/7 \\ -3 & -3/2 & 0 \end{bmatrix}$$

with the induced p=2 norm $||\mathbf{J}_g|| = 5.917 > 1$. The iteration does not converge.

Example 1.9 Consider a 3-variable nonlinear function $\mathbf{f}(\mathbf{x})$ of arguments $\mathbf{x} = [x,\ y,\ z]^T$:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \qquad \begin{cases} f_1(\mathbf{x}) = x^2 - 2x + y^2 - z + 1 = 0 \\ f_2(\mathbf{x}) = xy^2 - x - 3y + yz + 2 = 0 \\ f_3(\mathbf{x}) = xz^2 - 3z + yz^2 + xy = 0 \end{cases}$$

The g-function can be obtained as

$$\mathbf{g}(\mathbf{x}) = \mathbf{x}, \qquad \begin{cases} g_1(\mathbf{x}) = x = (x^2 + y^2 - z + 1)/2 \\ g_2(\mathbf{x}) = y = (xy^2 - x + yz + 2)/3 \\ g_3(\mathbf{x}) = z = (xz^2 + yz^2 + xy)/3 \end{cases}$$

With $\mathbf{x}_0 = [0,\ 0,\ 0]^T$ and after $n > 170$ iterations $\mathbf{x}_{n+1} = \mathbf{g}(\mathbf{x}_n)$ converges to $\mathbf{x}_n = [1.098933,\ 0.367621,\ 0.144932]^T$, with error $||\mathbf{f}(\mathbf{x}_n)|| < 10^{-7}$. However, the iteration may not converge from other possible initial guesses.

Finally, we consider Aitken's method, by which the iteration $x_{n+1} = g(x_n)$ can be accelerated based on two consecutive points x_0 and x_1, as shown in Fig. 1.9.

The secant line of $g(x)$ that goes through the two points $P_0 = (x_0, g(x_0) = x_1)$ and $P_1 = (x_1, g(x_1) = x_2)$ is represented by the equation in terms of its slope:

$$\frac{y - g(x_0)}{x - x_0} = \frac{g(x_1) - g(x_0)}{x_1 - x_0} \tag{1.36}$$

Solving for y, we get

$$y = g(x_0) + (x - x_0)\frac{g(x_1) - g(x_0)}{x_1 - x_0} = x_1 + (x - x_0)\frac{x_2 - x_1}{x_1 - x_0} \tag{1.37}$$

To accelerate, instead of moving from x_0 to x_1, we move to the point x at which this secant line intersects with the identity function $y = x$. We can therefore

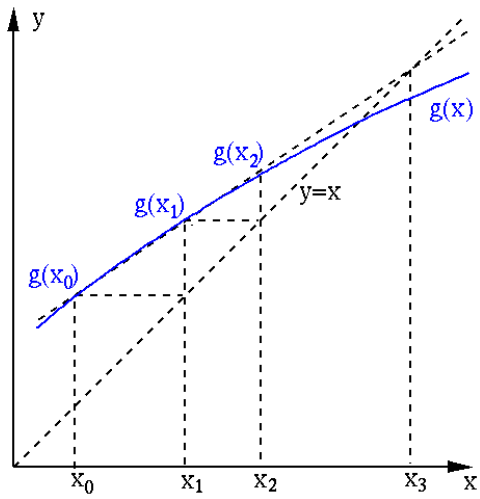


Figure 1.9 Aitken's Method

replace y in the equation above by x and solve the resulting equation

$$x = x_1 + (x - x_0)\frac{x_2 - x_1}{x_1 - x_0} \tag{1.38}$$

to get

$$x = \frac{x_1(x_1 - x_0) - x_0(x_2 - x_1)}{(x_1 - x_0) - (x_2 - x_1)} = x_0 - \frac{(x_1 - x_0)^2}{x_2 - 2x_1 + x_0} = x_0 - \frac{(\Delta x_0)^2}{\Delta^2 x_0} \tag{1.39}$$

where Δx_0 and $\Delta^2 x_0$ are respectively the first and second order differences defined below:

$$\Delta x_0 = x_1 - x_0, \qquad \Delta x_1 = x_2 - x_1 \tag{1.40}$$

$$\Delta^2 x_0 = \Delta x_1 - \Delta x_0 = (x_2 - x_1) - (x_1 - x_0) = x_2 - 2x_1 + x_0 \tag{1.41}$$

This result can then be converted into an iterative process

$$\begin{cases} x_{n+1} = g(x_n) \\ x_{n+2} = g(x_{n+1}) \\ x_{n+3} = x_n - (x_{n+1} - x_n)^2/(x_{n+2} - 2x_{n+1} + x_n) \end{cases} \tag{1.42}$$

Given x_n, we skip $x_{n+1} = g(x_n)$ and $x_{n+2} = g(x_{n+1})$ but directly move to x_{n+3} computed based on x_{n+1} and x_{n+2}, thereby making a greater step towards the solution.

Example 1.10 Solve $x^3 - 3x + 1 = 0$. Construct $g(x) = (3x - 1)^{1/3}$. It takes 18 iterations for the regular fixed-point algorithm with initial guess $x_0 = 1$, to get $x_{18} = 1.5320887$ that satisfies $|f(x_n)| < 10^{-6}$, but it only three iterations for Aitken's method to converge to the same result:

n	x_n	$f(x_n)$
0	1.0000	-1.0000
1	1.259921	$-7.797631e-01$
2	1.406056	$-4.384047e-01$
3	1.476396	$-2.110206e-01$
4	1.507985	$-9.476760e-02$
5	1.521751	$-4.129594e-02$
6	1.527672	$1.776362e-02$
7	1.530205	$-7.598914e-03$
8	1.531286	$3.242990e-03$
9	1.531747	$-1.382619e-03$
10	1.531943	$-5.892137e-04$
11	1.532027	$-2.510521e-04$
12	1.532062	$-1.069599e-04$
13	1.532078	$-4.556840e-05$
14	1.532084	$-1.941335e-05$
15	1.532087	$-8.270551e-06$
16	1.532088	$-3.523444e-06$
17	1.532089	$-1.501066e-06$
18	1.532089	$-6.394874e-07$

n	x_n	$f(x_n)$
0	1.0000	-1.0000
1	1.5937361	$2.668730e-01$
2	1.5323992	$1.254749e-03$
3	1.5320889	$3.421160e-08$

1.4 Newton-Raphson Method (Univariate)

The Newton-Raphson method, or simply Newton's method, solves the general equation $f(x) = 0$ more efficiently as its converges more quickly (quadratic) than the secant method previously discussed, due to the derivative $f'(x)$ assumed to be available, as an extra piece of information in additional to the function value $f(x)$ itself. Newton's method is based on the approximation of the function $f(x)$ by its Taylor series expansion at any x_0 as an initial guess of the root:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!}f''(x)(x - x_0)^2 + \cdots + \frac{1}{n!}f^{(n)}(x_0)(x - x_0)^n + \cdots \tag{1.43}$$

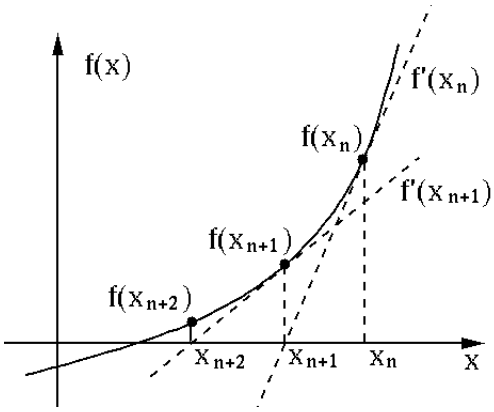


Figure 1.10 Newton-Raphson Method

If $f(x)$ is linear, i.e., its slope $f'(x)$ is a constant at any point x, then all terms other than the first two are zero, and the equation becomes

$$f(x) = f(x_0) + f'(x)(x - x_0) = 0 \tag{1.44}$$

Solving this equation for x we get the root x^*:

$$x^* = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 + \Delta x_0 \tag{1.45}$$

where $\Delta x_0 = -f(x_0)/f'(x_0)$ is the step increment needed to move from the initial guess x_0 to the root x^*:

$$\Delta x_0 = -\frac{f(x_0)}{f'(x_0)} \begin{cases} < 0 & \text{if } f(x_0) \text{ and } f'(x_0) \text{ are of the same sign} \\ > 0 & \text{if } f(x_0) \text{ and } f'(x_0) \text{ are of different signs} \end{cases} \tag{1.46}$$

If $f(x)$ is nonlinear, the sum of the first two terms of its Taylor series is only an approximation of the function, and the result in Eq. (1.45) is no longer the root, but an approximation of the root, which can be improved iteratively to gradually approach the root x^*:

$$x_{n+1} = x_n + \Delta x_n = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n = 0,\, 1,\, 2, \cdots \tag{1.47}$$

as illustrated in Fig. 1.10. Note that this iteration is the same as Eq. (1.20) for the secant method, with $\hat{f}'(x_n)$ replaced by the derivative $f'(x_n)$.

The Newton-Raphson method can be considered as the fixed-point iteration $x_{n+1} = g(x_n)$ based on

$$g(x) = x - f(x)/f'(x) \tag{1.48}$$

based on the assumption that $f'(x) \neq 0$. The root x^* at which $f(x^*) = 0$ is also the fixed point of $g(x)$, i.e., $g(x^*) = x^*$. For the iteration to converge, $g(x)$ needs

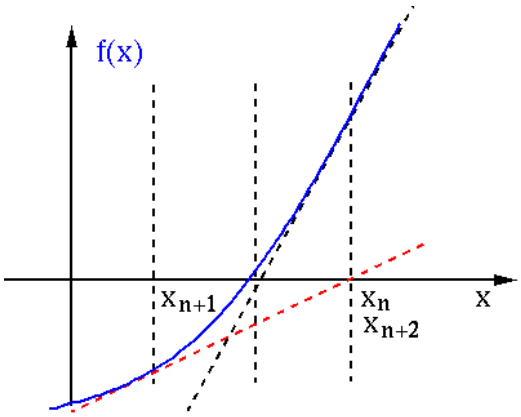


Figure 1.11 Oscillation

to be a contraction with $|g'(x)| < 1$. Consider

$$g'(x) = \left(x - \frac{f(x)}{f'(x)}\right)' = 1 - \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2} \tag{1.49}$$

At the root $x = x^*$ where $f(x^*) = 0$, if $f'(x^*) \neq 0$, then $g'(x^*) = 0 < 1$, i.e., $g(x)$ is a contraction and the iteration $x_{n+1} = g(x_n)$ converges quadratically when x_n is close enough to x^*.

Here are some additional considerations of the Newton-Raphson methods:

- If $f'(x_n) = 0$ (with a horizontal tangent line), the iteration cannot proceed, but we can modify x_n by adding a small value ϵ to x_n so that $f'(x+\epsilon) \neq 0$.
- It is difficult to know the number of roots of a nonlinear equation (unlike the case of a linear equation), which can vary from zero (e.g., $f(x) = x^2 + 1$) to infinity (e.g., $f(x) = \cos(x)$). One can try different initial guesses in the range of interest to see if different roots can be found.
- Sometime a parameter δ can be used to control the step size of the iteration:

$$x_{n+1} = x_n - \delta \frac{f(x_n)}{f'(x_n)} \tag{1.50}$$

  - If $\delta < 1$, the iteration is de-accelerated. Although the convergence becomes slower, this may be desirable if the function $f(x)$ is not smooth with many local variations.
  - If $\delta > 1$, the iteration is accelerated. The convergence may or may not be accelerated. Due to the greater step size, the root may be skipped and missed. Sometimes the convergence may become significantly slowed or even oscillate around the true root, such as the example shown in Fig. 1.11 with $\delta = 2$.

The Newton-Raphson method converges quadratically:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^2} \leq \mu \tag{1.51}$$

as shown in Appendix 2 of this chapter, i.e., it converges more quickly than both the bisection method ($q = 1$) and the secant method ($q = 1.618$).

Example 1.11

$$f(x) = x^3 - 4x^2 + 5x - 2 = (x-1)^2(x-2) = 0 \tag{1.52}$$

Obviously this equation has a repeated root $x = 1$ as well as a single root $x = 2$. We have

$$f'(x) = 3x^2 - 8x + 5 = (3x-5)(x-1), \qquad f''(x) = 6x - 8 \tag{1.53}$$

Note that at the root $x^* = 1$ we have $f'(x^*) = f'(1) = 0$. We further find:

$$g(x) = x - \frac{f(x)}{f'(x)} = x - \frac{(x-1)^2(x-2)}{(3x-5)(x-1)} = x - \frac{(x-1)(x-2)}{3x-5} \tag{1.54}$$

and

$$\begin{aligned} g'(x) &= \frac{f(x)f''(x)}{(f'(x))^2} = \frac{(x-1)^2(x-2)(6x-8)}{(3x-5)^2(x-1)^2} \\ &= \frac{(x-2)(6x-8)}{(3x-5)^2} = \begin{cases} 1/2 & x = 1 \\ 0 & x = 2 \end{cases} \end{aligned}$$

We therefore have

$$e_{n+1} = g'(x)e_n - \frac{1}{2}g''(x)e_n^2 + O(e_n^3) \overset{n\to\infty}{\Longrightarrow} \begin{cases} e_n/2 & x = 1 \\ -g''(x)e_n^2/2 & x = 2 \end{cases} \tag{1.55}$$

We see that the iteration converges quadratically to the single root $x = 2$, but only linearly to the repeated root $x = 1$.

We consider in general a function with a repeated root at $x = a$ of multiplicity k:

$$f(x) = (x-a)^k h(x) \tag{1.56}$$

its derivative is

$$f'(x) = k(x-a)^{k-1}h(x) + (x-a)^k h'(x) = (x-a)^{k-1}[kh(x) + (x-a)h'(x)] \tag{1.57}$$

As $f'(a) = 0$, the convergence of the Newton-Raphson method to $x = a$ is linear, rather than quadratic.

In such case, we can accelerate the iteration by using a step size $\delta = k > 1$:

$$g(x) = x - k\,\frac{f(x)}{f'(x)} \tag{1.58}$$

and

$$g'(x) = 1 - k\,\frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} = 1 - k + k\,\frac{f(x)f''(x)}{(f'(x))^2} \tag{1.59}$$

Now we show that this $g'(x)$ is zero at the repeated root $x = a$, therefore the convergence to this root is still quadratic.

We substitute $f(x) = (x-a)^k h(x)$, $f'(x) = k(x-a)^{k-1}[kh + (x-a)h']$, and

$$\begin{aligned} f''(x) &= [(x-a)^{k-1}[kh + (x-a)h']]' \\ &= 1 - k + k\,(k-1)(x-a)^{k-2}[kh + (x-a)h'] \\ &\quad +(x-a)^{k-1}[(k+1)h' + (x-a)h''] \end{aligned} \tag{1.60}$$

into the expression for $g'(x)$ above to get (after some algebra):

$$\begin{aligned} g'(x) &= 1 - k + k\frac{f(x)f''(x)}{(f'(x))^2} \\ &= 1 - k + k\,\frac{h(k-1)(kh + (x-a)h') + h(x-a)((k+1)h' + (x-a)h')}{(kh + (x-a)h')^2} \end{aligned} \tag{1.61}$$

At $x = a$, we get

$$g'(x)\Big|_{x=a} = 1 - k + k\frac{(k-1)kh^2}{(kh)^2} = 0 \tag{1.62}$$

i.e., the convergence to the repeated root at $x = a$ is no longer linear but quadratic. The difficulty, however, is that the multiplicity k of a root is unknown ahead of time. If $\delta > 1$ is used blindly some root may be skipped, and the iteration may oscillate around the real root.

1.5 Newton-Raphson Method (Multivariate)

The Newton-Raphson method discussed above for solving a single-variable equation $f(x) = 0$ can be generalized to the case of multivariate equation systems containing M equations of N variables in $\mathbf{x} = [x_1, \cdots, x_N]^T$:

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_M(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} f_1(x_1, \cdots, x_N) \\ \vdots \\ f_M(x_1, \cdots, x_N) \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{0} \tag{1.63}$$

Again, we first consider the Taylor series expansion (Eq. (A.391)) of each of the M functions in the eighborhood of an initial guess $\mathbf{x}_0 = [x_{01}, \cdots, x_{0N}]^T$:

$$\begin{aligned} f_m(\mathbf{x}) &= f_m(\mathbf{x}_0) + \mathbf{g}_{f_m}^T(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + r_m(||\mathbf{x} - \mathbf{x}_0||^2) \\ &\approx f_m(\mathbf{x}_0) + \sum_{n=1}^{N} \frac{\partial f_m(\mathbf{x}_0)}{\partial x_n}(x_n - x_{0n}) \qquad (m = 1, \cdots, M) \end{aligned} \tag{1.64}$$

where $r_m(||\mathbf{x}-\mathbf{x}_0||^2)$ represents the second and higher order terms in the series beyond the linear term, which can be neglected if $||\mathbf{x}-\mathbf{x}_0||$ is small. These M equations can be expressed in matrix form

$$\begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_M(\mathbf{x}) \end{bmatrix} \approx \begin{bmatrix} f_1(\mathbf{x}_0) \\ \vdots \\ f_M(\mathbf{x}_0) \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1(\mathbf{x}_0)}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x}_0)}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M(\mathbf{x}_0)}{\partial x_1} & \cdots & \frac{\partial f_M(\mathbf{x}_0)}{\partial x_N} \end{bmatrix} \begin{bmatrix} x_1 - x_{01} \\ \vdots \\ x_N - x_{0N} \end{bmatrix} \tag{1.65}$$

or more concisely

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{J}(\mathbf{x}_0)\,(\mathbf{x}-\mathbf{x}_0) = \mathbf{f}_0 + \mathbf{J}_0\,\Delta\mathbf{x} \tag{1.66}$$

where $\Delta\mathbf{x} = \mathbf{x}-\mathbf{x}_0$, and $\mathbf{f}_0 = \mathbf{f}(\mathbf{x}_0)$ and $\mathbf{J}_0 = \mathbf{J}(\mathbf{x}_0)$ are respectively the function $\mathbf{f}(\mathbf{x})$ and its *Jacobian matrix* $\mathbf{J}_f(\mathbf{x})$ both evaluated at $\mathbf{x}_0$. We further consider solving the equation system $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ in the following two cases:

- $M = N$: The number of equations is the same as the number of unknowns, the Jacobian $\mathbf{J}(\mathbf{x})$ is a square matrix and its inverse $\mathbf{J}^{-1}$ exists in general. In the special case where $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$ is linear, the Taylor series contains only the first two terms while all higher order terms are zero, and the approximation in Eq. (1.66) becomes exact with $\mathbf{J} = \mathbf{A}$. To find the root $\mathbf{x}^*$ of the equation, we set $\mathbf{f}(\mathbf{x})$ in the equation to zero and solve the resulting equation for $\mathbf{x}$ to get the solution:

$$\mathbf{x}^* = \mathbf{x}_0 - \mathbf{J}_0^{-1}\mathbf{f}_0 = \mathbf{x}_0 - \mathbf{A}^{-1}\mathbf{f}_0 \tag{1.67}$$

  For example, if $\mathbf{x}_0 = \mathbf{0}$ and $\mathbf{f}(\mathbf{x}_0) = -\mathbf{b}$, then $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$.

  If $\mathbf{f}(\mathbf{x})$ is nonlinear, the sum of the first two terms of the Taylor series is only an approximation, and the result in Eq. (1.67) is only an approximate root, which can be further improved iteratively to move from the initial guess $\mathbf{x}_0$ towards the root $\mathbf{x}^*$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_n = \mathbf{x}_n - \mathbf{J}_n^{-1}\,\mathbf{f}_n \tag{1.68}$$

  where $\Delta\mathbf{x}_n = -\mathbf{J}_n^{-1}\,\mathbf{f}_n$ is the increment, representing the *search direction* at the nth step. The iteration moves $\mathbf{x}_n$ in the N-D space spanned by $\{x_1, \cdots, x_N\}$ from some initial guess $\mathbf{x}_0$ along such a path that all function values $f_m(\mathbf{x}),\quad m = 1, \cdots, M)$ are reduced. As in the univariate case, a scaling factor δ_n can be used to control the step size of the iteration

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \delta_n\,\Delta\mathbf{x}_n = \mathbf{x}_n - \delta_n\,\mathbf{J}_n^{-1}\,\mathbf{f}_n \tag{1.69}$$

  The step size can be adjusted based on the specific function shape. When $\delta_n < 1$, the step size becomes smaller and the convergence of the iteration is slower, we will have a better chance not to skip a solution, which may happen if $\mathbf{f}(\mathbf{x})$ is not smooth and the step size is too big.

  The algorithm is listed below (where the tolerance *tol* is preset to a small value):

  - Select $\mathbf{x}_0$
  - Obtain $\mathbf{f}_0 = \mathbf{f}(\mathbf{x}_0)$ and $\mathbf{J}_0$
  - Obtain $\mathbf{J}_0^{-1}$
  - $n = 0$
  - While $||\mathbf{x}_{n+1} - \mathbf{x}_n|| > tol$ do
    - $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_n^{-1}\mathbf{f}_n$
    - Find $\mathbf{f}_{n+1}$ and $\mathbf{J}_{n+1}$
    - $n = n + 1$

- $M > N$: There are more equations than unknowns, i.e., equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ is an over-constrained system, and the Jacobian $\mathbf{J}(\mathbf{x})$ is an $M \times N$ non-square matrix with no inverse, i.e., no solution exists for the equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ in general. But we can still seek to find an optimal solution $\mathbf{x}^*$ that minimizes the following sum-of-squares error (SSE):

$$\varepsilon(\mathbf{x}) = \frac{1}{2}||\mathbf{f}(\mathbf{x})||^2 = \frac{1}{2}\mathbf{f}(\mathbf{x})^T\mathbf{f}(\mathbf{x}) = \frac{1}{2}\sum_{m=1}^{M} f_m^2(\mathbf{x}) \tag{1.70}$$

The gradient vector of this SSE is

$$\begin{aligned}\mathbf{g}_\varepsilon(\mathbf{x}) &= \frac{d}{d\mathbf{x}}\varepsilon(\mathbf{x}) = \frac{d}{d\mathbf{x}}\left(\frac{1}{2}\sum_{m=1}^{M} f_m^2(\mathbf{x})\right) = \sum_{m=1}^{M}\frac{d\, f_m(\mathbf{x})}{d\mathbf{x}} f_m(\mathbf{x}) \\ &= \mathbf{J}_f^T(\mathbf{x})\,\mathbf{f}(\mathbf{x})\end{aligned} \tag{1.71}$$

where $\mathbf{J}_f(\mathbf{x})$ is the Jacobian of $\mathbf{f}(\mathbf{x})$. We can also obtain the nth component $(n = 1, \cdots, N)$ of $\mathbf{g}_\varepsilon(\mathbf{x})$ as:

$$\frac{\partial\varepsilon(\mathbf{x})}{\partial x_n} = \sum_{m=1}^{M}\frac{\partial f_m(\mathbf{x})}{\partial x_n}\, f_m(\mathbf{x}) = \sum_{m=1}^{M} J_{mn}\, f_m(\mathbf{x}) \tag{1.72}$$

where $J_{mn} = \partial f_m(\mathbf{x})/\partial x_n$ is the component in the mth row and nth column of $\mathbf{J}_f(\mathbf{x})$. At the optimal solution $\mathbf{x}^*$ that minimizes this error $\varepsilon(\mathbf{x})$, we have $\mathbf{g}_\varepsilon(\mathbf{x}^*) = \mathbf{0}$.

In the special case when the function $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$ is linear with $\mathbf{J}(\mathbf{x}) = \mathbf{A}$, it can be represented as the sum of the first two terms of its Taylor series in Eq. (1.66), and the gradient is:

$$\mathbf{g}_\varepsilon(\mathbf{x}) = \mathbf{A}^T\,\mathbf{f}(\mathbf{x}) = \mathbf{A}^T\,[\mathbf{f}(\mathbf{x}_0) + \mathbf{A}\,\Delta\mathbf{x}] = \mathbf{A}^T(\mathbf{f}_0 + \mathbf{A}\Delta\mathbf{x}) \tag{1.73}$$

where $\mathbf{x}_0$ is any chosen initial guess. At the optimal solution $\mathbf{x}^*$ that minimizes $\varepsilon(\mathbf{x})$, this gradient is zero:

$$\mathbf{g}_\varepsilon(\mathbf{x}^*) = \mathbf{A}^T(\mathbf{f}_0 + \mathbf{A}\Delta\mathbf{x}) = \mathbf{0} \tag{1.74}$$

Solving this equation for $\Delta\mathbf{x}$ we get the optimal increment

$$\Delta\mathbf{x} = -(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{f}_0 = -\mathbf{A}^-\,\mathbf{f}_0 \tag{1.75}$$

Here $\mathbf{A}^- = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ is the *pseudo-inverse* (Section A.6.1) of the non-square matrix $\mathbf{A}$. Now the optimal solution can be found as:

$$\mathbf{x}^* = \mathbf{x}_0 + \Delta\mathbf{x} = \mathbf{x}_0 - \mathbf{A}^- \mathbf{f}_0 \tag{1.76}$$

For example, if $\mathbf{x}_0 = \mathbf{0}$ and $\mathbf{f}(\mathbf{x}_0) = \mathbf{A}\mathbf{x}_0 - \mathbf{b} = -\mathbf{b}$, we have $\mathbf{x}^* = \mathbf{A}^-\mathbf{b}$, same as Eq. (1.8).

In the general case when $\mathbf{f}(\mathbf{x})$ is nonlinear, the sum of the first two terms of its Taylor series is its approximation, and the result above is an approximation of the root, which can be further improved iteratively to move from the initial guess towards the root:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_n = \mathbf{x}_n - \mathbf{J}_n^- \mathbf{f}_n \tag{1.77}$$

This iteration will converge to $\mathbf{x}^*$ at which $\mathbf{g}_\varepsilon(\mathbf{x}^*) = \mathbf{0}$, and the squared error $\varepsilon(\mathbf{x})$ is minimized.

Comparing Eqs. (1.68) and (1.77), we see that they are essentially the same, with the only difference that the regular inverse $\mathbf{J}^{-1}$ is used when $M = N$, but the pseudoinverse $\mathbf{J}^-$ is used when $M > N$ and $\mathbf{J}^{-1}$ does not exist.

The Newton-Raphson method assumes the availability of the analytical expressions of all partial derivatives $J_{mn} = \partial f_m(\mathbf{x})/\partial x_n$ $(m = 1, \cdots, M,\ n = 1, \cdots, N)$ in the Jacobian matrix $\mathbf{J}$. However, when this is not the case, J_{mn} need to be approximated by forward or central difference (secant) method:

$$\begin{aligned} J_{mn} = \frac{\partial f_m(x_1, \cdots, x_N)}{\partial x_n} &\approx \frac{f_m(x_1, \cdots, x_n + h, \cdots, x_N) - f_m(x_1, \cdots, x_n, \cdots, x_N)}{h} \\ &\approx \frac{f_m(x_1, \cdots, x_n + h, \cdots, x_N) - f_m(x_1, \cdots, x_n - h, \cdots, x_N)}{2h} \end{aligned} \tag{1.78}$$

where h is a small increment.

Example 1.12

$$\begin{cases} 3\,x_1 - \cos(x_2 x_3) - 3/2 = 0 \\ 4x_1^2 - 625\,x_2^2 + 2x_3 - 1 = 0 \\ 20\,x_3 + e^{-x_1 x_2} + 9 = 0 \end{cases}$$

$$\mathbf{J} = \begin{bmatrix} 3 & x_3 \sin(x_2 x_3) & x_2 \sin(x_2 x_3) \\ 8\,x_1 & -1250\,x_2 & 2 \\ -x_2\, e^{-x_1 x_2} & -x_1\, e^{-x_1 x_2} & 20 \end{bmatrix}$$

n	$\mathbf{x}$	$error$
0	$(1.000000, 1.000000, 1.000000)$	$6.207e+02$
1	$(1.232701, 0.503132, -0.473253)$	$1.541e+02$
2	$(0.832592, 0.251806, -0.490636)$	$3.884e+01$
3	$(0.833238, 0.128406, -0.494702)$	$9.517e+00$
4	$(0.833275, 0.069082, -0.497147)$	$2.200e+00$
5	$(0.833281, 0.043585, -0.498206)$	$4.063e-01$
6	$(0.833282, 0.036117, -0.498517)$	$3.486e-02$
7	$(0.833282, 0.035343, -0.498549)$	$3.741e-04$
8	$(0.833282, 0.035335, -0.498549)$	$4.498e-08$
9	$(0.833282, 0.035335, -0.498549)$	$5.551e-16$

Example 1.13

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \qquad \begin{cases} f_1(\mathbf{x}) = x_1^2 - 2x_1 + x_2^2 - x_3 + 1 = 0 \\ f_2(\mathbf{x}) = x_1 x_2^2 - x_1 - 3x_2 + x_2 x_3 + 2 = 0 \\ f_3(\mathbf{x}) = x_1 x_3^2 - 3x_3 + x_2 x_3^2 + x_1 x_2 = 0 \end{cases}$$

$$\mathbf{J} = \begin{bmatrix} 2x_1 - 2 & 2x_2 & -1 \\ x_2^2 - 1 & 2x_1 x_2 - 3 + x_3 & x_2 \\ x_3^2 + x_2 & x_3^2 + x_1 & 2x_1 x_3 - 3 + 2x_2 x_3 \end{bmatrix}$$

With $\mathbf{x}_0 = [1,\ 2,\ 3]^T$, we get a root:

n	$\mathbf{x}$	$error$
0	$(1.00000, 2.00000, 3.00000)$	$2.064e+01$
1	$(0.10256, 1.64103, 2.56410)$	$4.303e+00$
2	$(1.52062, 1.41113, 0.19859)$	$2.689e+00$
3	$(1.94123, 0.77134, 0.89465)$	$1.217e+00$
4	$(1.06737, 1.19117, 0.48353)$	$1.144e+00$
5	$(1.26825, 0.95182, 0.88028)$	$3.323e-01$
6	$(0.95899, 1.03384, 0.96813)$	$1.171e-01$
7	$(1.00171, 1.00007, 0.99718)$	$4.162e-03$
8	$(1.00000, 1.00000, 1.00000)$	$6.701e-06$

With $\mathbf{x}_0 = [0,\ 0,\ 0]^T$, we get another root:

n	$\mathbf{x}$	*error*
0	$(0.00000, 0.00000, 0.00000)$	$2.236e+00$
1	$(0.50000, 0.50000, 0.00000)$	$5.728e-01$
2	$(0.83951, 0.47531, 0.13580)$	$1.175e-01$
3	$(0.98582, 0.41849, 0.15069)$	$2.639e-02$
4	$(1.05417, 0.38715, 0.14717)$	$6.088e-03$
5	$(1.08565, 0.37339, 0.14558)$	$1.264e-03$
6	$(1.09693, 0.36849, 0.14503)$	$1.618e-04$
7	$(1.09888, 0.36764, 0.14494)$	$4.817e-06$

Broyden's method

In the Newton-Raphson method, two main operations are carried out in each iteration: (a) evaluate the Jacobian matrix $\mathbf{J_f}(\mathbf{x}_n)$ and (b) obtain its inverse $\mathbf{J}_\mathbf{f}^{-1}(\mathbf{x}_n)$. To avoid the expensive computation for these operations, we can consider using Broyden's method, one of the so called *quasi-Newton methods*, which approximates the inverse of the Jacobian $\mathbf{J}_{n+1}^{-1} = \mathbf{J}^{-1}(\mathbf{x}_{n+1})$ from the $\mathbf{J}_n^{-1} = \mathbf{J}^{-1}(\mathbf{x}_n)$ in the previous iteration step, so that it can be updated iteratively from the initial $\mathbf{J}_0^{-1} = \mathbf{J}^{-1}(\mathbf{x}_0)$.

We first consider the iteration step in the single-variable case to see how to estimate the next derivative $f'_{n+1} = f'(x_{n+1})$ from the current $f'_n = f'(x_n)$ by the secant method:

$$\begin{aligned} f'_{n+1} \approx \hat{f}'_{n+1} &= \frac{f_{n+1} - f_n}{x_{n+1} - x_n} = \frac{\delta f_n}{\delta x_n} = \frac{\hat{f}'_n \delta x_n - \hat{f}'_n \delta x_n + \delta f_n}{\delta x_n} \\ &= \hat{f}'_n + \frac{\delta f_n - \hat{f}'_n \delta x_n}{\delta x_n} = \hat{f}'_n + \hat{\delta} f'_n \end{aligned} \tag{1.79}$$

where

- $\delta f_n = f_{n+1} - f_n$ is the true increment of the function over the interval $\delta x_n = x_{n+1} - x_n$;
- $\hat{f}'_n \delta x_n$ is the estimated increment of the function based on the previous derivative $\hat{f}'_n$;
- $\hat{\delta} f'_n$ is the estimated increment of the derivative:

$$\hat{\delta} f'_n = \frac{\delta f_n - \hat{f}'_n \delta x_n}{\delta x_n} \tag{1.80}$$

The equation above indicates that the derivative f'_{n+1} in the $(n+1)th$ step can be estimated by adding the estimated increment $\hat{\delta} f'_n$ to the derivative $\hat{f}'n$ in the current nth step, as shown in Fig. refBroyden1.

Having obtained $\hat{f}'_{n+1}$, we can use the same iteration in the Newton-Raphson

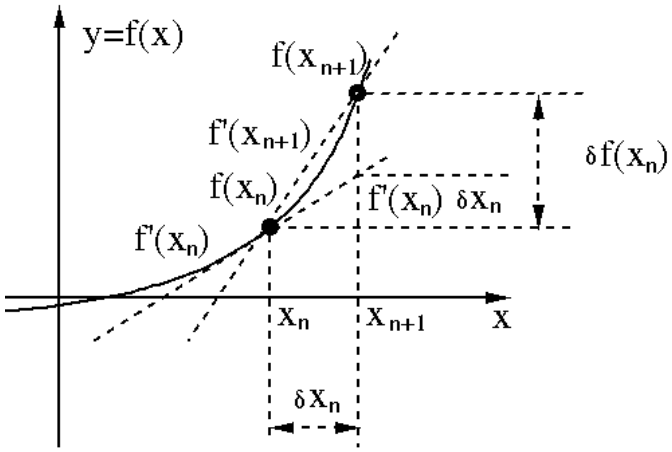


Figure 1.12 Broyden's Method (Section 1.5)

method to find x_{n+1}:

$$x_{n+1} = x_n - \frac{f(x_n)}{\hat{f}'_n} \tag{1.81}$$

This method for single-variable case can be generalized to multiple variable case for solving $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. Following the way we estimate the increment of the derivative of a single-variable function in Eq. (1.80), here we can estimate the increment of the Jacobian of a multi-variable function:

$$\delta\hat{\mathbf{J}}_n = \frac{(\delta\mathbf{f}_n - \hat{\mathbf{J}}_n\delta\mathbf{x}_n)\ \delta\mathbf{x}_n^T}{\delta\mathbf{x}_n^T\ \delta\mathbf{x}_n} = \frac{(\delta\mathbf{f}_n - \hat{\mathbf{J}}_n\delta\mathbf{x}_n)\delta\mathbf{x}_n^T}{||\delta\mathbf{x}_n||^2} \tag{1.82}$$

where $\delta\mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n$ and $\delta\mathbf{f}_n = \mathbf{f}_{n+1} - \mathbf{f}_n = \mathbf{f}(\mathbf{x}_{n+1}) - \mathbf{f}(\mathbf{x}_n)$. Now in each iteration, we can update the estimated Jacobian as well as the estimated root:

$$\hat{\mathbf{J}}_{n+1} = \hat{\mathbf{J}}_n + \delta\hat{\mathbf{J}}_n, \qquad \mathbf{x}_{n+1} = \mathbf{x}_n + \delta\mathbf{x}_n = \mathbf{x}_n - \hat{\mathbf{J}}_n^{-1}\mathbf{f}(\mathbf{x}_n) \tag{1.83}$$

The algorithm can be further improved so that the inverse of the Jacobian $\mathbf{J}_n$ is avoided. Specifically, consider the inverse Jacobian:

$$\hat{\mathbf{J}}_{n+1}^{-1} = \left(\hat{\mathbf{J}}_n + \delta\hat{\mathbf{J}}_n\right)^{-1} = \left[\hat{\mathbf{J}}_n + \frac{(\delta\mathbf{f}_n - \hat{\mathbf{J}}_n\delta\mathbf{x}_n)\delta\mathbf{x}_n^T}{||\delta\mathbf{x}_n||^2}\right]^{-1} \tag{1.84}$$

We can apply the Sherman-Morrison formula (Section A.2.4):

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T\mathbf{A}^{-1}}{1 + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}} \tag{1.85}$$

to the right-hand side of the equation above by defining

$$\mathbf{A} = \hat{\mathbf{J}}_n, \qquad \mathbf{u} = \frac{\delta\mathbf{f}_n - \hat{\mathbf{J}}_n\delta\mathbf{x}_n}{||\delta\mathbf{x}_n||^2}, \qquad \mathbf{v} = \delta\mathbf{x}_n \tag{1.86}$$

and rewrite it as:

$$\hat{\mathbf{J}}_{n+1}^{-1} = \hat{\mathbf{J}}_n^{-1} - \frac{\hat{\mathbf{J}}_n^{-1}\left(\delta\mathbf{f}_n - \hat{\mathbf{J}}_n\delta\mathbf{x}_n^T\right)/||\delta\mathbf{x}_n||^2\,\delta\mathbf{x}_n^T\,\hat{\mathbf{J}}_n^{-1}}{1+\delta\mathbf{x}_n^T\hat{\mathbf{J}}_n^{-1}(\delta\mathbf{f}_n - \hat{\mathbf{J}}_n\delta\mathbf{x}_n)/||\delta\mathbf{x}_n||^2} = \hat{\mathbf{J}}_n^{-1} - \frac{\left(\hat{\mathbf{J}}_n^{-1}\delta\mathbf{f}_n - \delta\mathbf{x}_n\right)\delta\mathbf{x}_n^T\,\hat{\mathbf{J}}_n^{-1}}{\delta\mathbf{x}_n^T\hat{\mathbf{J}}_n^{-1}\delta\mathbf{f}_n} \tag{1.87}$$

We see that the next $\hat{\mathbf{J}}_{n+1}^{-1}$ can be iteratively estimated directly from the previous $\hat{\mathbf{J}}_n^{-1}$, thereby avoiding computing the inverse of $\hat{\mathbf{J}}_n$ altogether. The algorithm is listed below:

- Select $\mathbf{x}_0$
- Find $\mathbf{f}_0 = \mathbf{f}(\mathbf{x}_0)$, $\mathbf{J}_0 = \mathbf{J}(\mathbf{x}_0)$, and $\mathbf{J}_0^{-1}$
- $\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{J}_0^{-1}\mathbf{f}_0$
- $n = 0$
- While $||\mathbf{x}_{n+1} - \mathbf{x}_n|| > tol$ do
  - $\delta\mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n$
  - $\mathbf{f}_{n+1} = \mathbf{f}(\mathbf{x}_{n+1})$
  - $\delta\mathbf{f} = \mathbf{f}_{n+1} - \mathbf{f}_n$
  - $\mathbf{J}_n^{-1} = \mathbf{J}_n^{-1} - (\delta\mathbf{x}_n - \mathbf{J}_n^{-1}\delta\mathbf{f}_n)\delta\mathbf{x}_n^T\mathbf{J}_n^{-1}/(\delta\mathbf{x}_n^T\mathbf{J}_n^{-1}\delta\mathbf{f}_n)$
  - $\mathbf{x}_n = \mathbf{x}_{n+1}$, $\mathbf{f}_n = \mathbf{f}_{n+1}$
  - $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_n^{-1}\mathbf{f}_n$
  - $n = n + 1$

Problems

Develop Matlab code (or any other language) to implement the algorithms covered in this chapter to solve the equations in the examples in the text. Try a set of different initial guesses to see how they may affect how quickly the iteration converges.

1. Implement the bisection and secant methods and apply them to solve the single-variable equations considered in Examples 1.2 through 1.6 based on properly selected initial guesses.
2. Implement method of fixed-point iteration and apply it to solve the same single-variable equations in the previous problem based on properly selected initial guesses.
3. Implement the univariate Newton-Raphson method and apply it to solve the same single-variable equations above.
4. Implement the multivariate Newton-Raphson method and apply it to solve the multivariable equations in Examples 1.12 and 1.13.

Appendices

1.A Order of Convergence of the Secant Method

Let x^* be the root at which $f(x^*) = 0$. The error of x_{n+1} is:

$$\begin{aligned} e_{n+1} = x_{n+1} - x^* &= x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n) - x^* \\ &= \frac{(x_{n-1} - x^*)f(x_n) - (x_n - x^*)f(x_{n-1})}{f(x_n) - f(x_{n-1})} \\ &= \frac{e_{n-1}f(x_n) - e_n f(x_{n-1})}{f(x_n) - f(x_{n-1})} \end{aligned} \tag{1.88}$$

Consider the Taylor expansion of $f(x_n)$ around the root at which as $f(x^*) = 0$:

$$\begin{aligned} f(x_n) = f(x^* + e_n) &= f(x^*) + f'(x^*)e_n + \frac{1}{2}f''(x^*)e_n^2 + O(e_n^3) \\ &= f'(x^*)e_n + \frac{1}{2}f''(x^*)e_n^2 + O(e_n^3) \end{aligned} \tag{1.89}$$

Similarly we also have

$$f(x_{n-1}) = f'(x^*)e_{n-1} + \frac{1}{2}f''(x^*)e_{n-1}^2 + O(e_{n-1}^3) \tag{1.90}$$

Substituting these into the expression for e_{n+1} above we get

$$\begin{aligned} e_{n+1} &= \frac{e_{n-1}[f'(x^*)e_n + \frac{1}{2}f''(x^*)e_n^2 + O(e_n^3)] - e_n[f'(x^*)e_{n-1} + \frac{1}{2}f''(x^*)e_{n-1}^2 + O(e_{n-1}^3)]}{[f'(x^*)e_n + \frac{1}{2}f''(x^*)e_n^2 + O(e_n^3)] - [f'(x^*)e_{n-1} + \frac{1}{2}f''(x^*)e_{n-1}^2 + O(e_{n-1}^3)]} \\ &= \frac{e_{n-1}e_n f''(x^*)(e_n - e_{n-1})/2 + O(e_{n-1}^4)}{(e_n - e_{n-1})f'(x^*) + (e_n^2 - e_{n+1}^2)f''(x^*)/2 + O(e_{n-1}^3)} \\ &= \frac{e_{n-1}e_n f''(x^*)/2 + O(e_{n-1}^3)}{f'(x^*) + (e_n + e_{n+1})f''(x^*)/2 + O(e_{n-1}^2)} \\ &= \frac{e_{n-1}e_n f''(x^*)/2 + O(e_{n-1}^3)}{f'(x^*) + O(e_n) + O(e_{n-1}^2)} \end{aligned} \tag{1.91}$$

When $n \to \infty$, the lowest order terms in both the numerator and denominator become the dominant terms as all other higher order terms approach to zero, and we have

$$e_{n+1} = \frac{e_{n-1}e_n f''(x^*)}{2f'(x^*)} = C\, e_n e_{n-1} \tag{1.92}$$

where we have defined a constant $C = f''(x^*)/2f'(x^*)$. To find the order of convergence, we need to find q in

$$|e_{n+1}| \leq |C|\ |e_{n-1}|\ |e_n| = \mu |e_n|^q \tag{1.93}$$

Solving this equation for $|e_n|$ we get

$$|e_n| = \left(\frac{|C|}{\mu}|e_{n-1}|\right)^{1/(q-1)} = \left(\frac{|C|}{\mu}\right)^{1/(q-1)} |e_{n-1}|^{1/(q-1)} \tag{1.94}$$

On the other hand, when $n \to \infty$ we also have

$$|e_n| = \mu |e_{n-1}|^q \tag{1.95}$$

Equating the right-hand sides of the two equations above we get

$$\left(\frac{|C|}{\mu}\right)^{1/(q-1)} |e_{n-1}|^{1/(q-1)} = \mu |e_{n-1}|^q \tag{1.96}$$

which requires the following two equations to hold:

$$q = \frac{1}{q-1}, \qquad \mu = \left(\frac{|C|}{\mu}\right)^{1/(q-1)} = \left(\frac{|C|}{\mu}\right)^q \tag{1.97}$$

These two equations can be solved separately to get

$$q = \frac{1+\sqrt{5}}{2} = 1.618, \qquad \mu = |C|^{q/(q+1)} = C^{(\sqrt{5}-1)/2} = |C|^{0.618} \tag{1.98}$$

i.e.,

$$|e_{n+1}| = \mu |e_n|^q = |C|^{0.618} |e_n|^{1.618} = \left|\frac{f''(x)}{2f'(x)}\right|^{0.618} |e_n|^{1.618} \tag{1.99}$$

or

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^{1.618}} = \mu = |C|^{0.618} \tag{1.100}$$

We see that the secant method has an order of convergence $q = 1.618$ with rate of convergence $\mu = |C|^{0.618}$.

1.B Order of Convergence of the Newton-Raphson Method

The order of convergence of the Newton-Raphson iteration can be found based on the Taylor expansion of $f(x)$ at the neighborhood of the root $x^* = x_n + e_n$ (Section A.7):

$$0 = f(x^*) = f(x_n) + f'(x_n)e_n + \frac{f''(x_n)}{2} e_n^2 + O(e_n^3) \tag{1.101}$$

where $e_n = x^* - x_n$ is the error at the nth step. Substituting the Newton-Raphson's iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad \text{i.e.} \quad f(x_n) = f'(x_n)(x_n - x_{n+1}) \tag{1.102}$$

into the equation above, we get

$$\begin{aligned} 0 &= f'(x_n)(x_n - x_{n+1}) + f'(x_n)(x^* - x_n) + \frac{f''(x_n)}{2} e_n^2 + O(e_n^3) \\ &= f'(x_n)(x^* - x_{n+1}) + \frac{f''(x)}{2} e_n^2 + O(e_n^3) \\ &= f'(x_n) e_{n+1} + \frac{f''(x_n)}{2} e_n^2 + O(e_n^3) \end{aligned} \tag{1.103}$$

i.e.

$$e_{n+1} = -\frac{f''(x_n)}{2f'(x_n)}e_n^2 + O(e_n^3) \tag{1.104}$$

When $n \to \infty$ all the higher order terms disappear, and the above can be written as

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^2} \leq \frac{|f''(x^*)|}{2|f'(x^*)|} = \mu \tag{1.105}$$

Alternatively, we can get the Taylor expansion in terms of $g(x)$:

$$x_{n+1} = x^* - e_{n+1} = g(x_n) = g(x^* - e_n) = g(x^*) - g'(x^*)e_n + \frac{g''(x^*)}{2}e_n^2 + O(e_n^3) \tag{1.106}$$

Subtracting $g(x^*) = x^*$ from both sides we get:

$$e_{n+1} = g'(x^*)e_n - \frac{g''(x^*)}{2}e_n^2 + O(e_n^3) \tag{1.107}$$

Now we find $g'(x)$ and $g''(x)$:

$$g'(x) = \left(x - \frac{f(x)}{f'(x)}\right)' = 1 - \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2} \tag{1.108}$$

and

$$g''(x) = \left(\frac{f(x)f''(x)}{(f'(x))^2}\right)' = \frac{(f'(x)f''(x) + f(x)f'''(x))(f'(x))^2 - f(x)f''(x)2f'(x)f''(x)}{(f'(x))^4}$$
$$= \frac{(f'(x))^3 f''(x)}{(f'(x))^4} = \frac{f''(x)}{f'(x)} \tag{1.109}$$

Evaluating these at $x = x^*$ at which $f(x^*) = 0$, and substituting them back into the expression for e_{n+1} above, we see that the linear term is zero as $g'(x^*) = 0$, and get the same result as in Eq. (1.105):

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^2} \leq \frac{|f''(x^*)|}{2|f'(x^*)|} = \frac{|g''(x^*)|}{2} \tag{1.110}$$

We see that if $f'(x^*) \neq 0$, the order of convergence of the Newton-Raphson method is $q = 2$ with the rate of convergence $\mu = |f''(x)|/2|f'(x)|$. However, if $f'(x^*) = 0$, the convergence is linear rather than quadratic, as shown in some of the examples.

1.C Proofs of the Fixed Point and Contraction Mapping Theorems

Proof of the fixed point theorem:

Here the distance $d(\mathbf{x}-\mathbf{y})$ is specifically defined as the *p-norm* (section A.4.1) of the vector $\mathbf{x} - \mathbf{y}$:

$$d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_p = \left(\sum_{n=1}^{N}(x_n - y_n)^p\right)^{1/p} \tag{1.111}$$

where $p \geq 1$, e.g., $p = 1, 2, \infty$. For convenience, we can drop p so that $d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||$.

- We first prove constructively the existence of a fixed point. As $\mathbf{g}(\mathbf{x})$ is a contraction, we have

$$||\mathbf{x}_{n+1} - \mathbf{x}_n|| = ||\mathbf{g}(\mathbf{x}_n) - \mathbf{g}(\mathbf{x}_{n-1})|| \leq k||\mathbf{x}_n - \mathbf{x}_{n-1}|| \leq k^2||\mathbf{x}_{n-1} - \mathbf{x}_{n-2}|| \leq \cdots \leq k^n||\mathbf{x}_1 - \mathbf{x}_0|| \tag{1.112}$$

  As $0 \leq k < 1$, we have $\lim_{n\to\infty} ||\mathbf{x}_{n+1} - \mathbf{x}_n|| = 0$. This is a *Cauchy sequence* (Section A.1) that converges to some point $\lim_{n\to\infty} \mathbf{x}_n = \mathbf{x}^*$ also in the space. We further have

$$\mathbf{g}(\mathbf{x}^*) = \mathbf{g}(\lim_{n\to\infty} \mathbf{x}_n) = \lim_{n\to\infty} \mathbf{g}(\mathbf{x}_n) = \lim_{n\to\infty} \mathbf{x}_{n+1} = \mathbf{x}^* \tag{1.113}$$

  i.e., the limit of the Cauchy sequence $\lim_{n\to\infty} \mathbf{x}_n = x^*$ is a fixed point.
- We next prove the uniqueness of the fixed point. Let $\mathbf{x}_1^*$ and $\mathbf{x}_2^*$ be two fixed points of $\mathbf{g}(\mathbf{x})$, then we have

$$||\mathbf{g}(\mathbf{x}_1^*) - \mathbf{g}(\mathbf{x}_2^*)|| \leq k||\mathbf{x}_1^* - \mathbf{x}_2^*|| = k||\mathbf{g}(\mathbf{x}_1^*) - \mathbf{g}(\mathbf{x}_2^*)|| \tag{1.114}$$

  For any $k \neq 0$, the above holds only if $||\mathbf{x}_1^* - \mathbf{x}_2^*|| = ||\mathbf{g}(\mathbf{x}_1^*) - f(\mathbf{x}_2^*)|| = 0$, i.e., $\mathbf{x}_1^* = \mathbf{x}_2^*$ is the unique fixed point.

QED

Proof of the Contraction Mapping theorem:

Consider the Taylor expansion (Section A.7) of the function $\mathbf{g}(\mathbf{x})$ in the neighborhood of $\mathbf{x}^*$:

$$\mathbf{g}(\mathbf{x}) = \mathbf{g}(\mathbf{x}^*) + \mathbf{g}'(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*) + R(\mathbf{x} - \mathbf{x}^*) = \mathbf{g}(\mathbf{x}^*) + \mathbf{J}_\mathbf{g}(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*) + R(\mathbf{x} - \mathbf{x}^*) \tag{1.115}$$

where $R(\mathbf{x} - \mathbf{x}^*)$ is the remainder composed of second and higher order terms of $\boldsymbol{\delta} = \mathbf{x} - \mathbf{x}^*$. Subtracting $\mathbf{g}(\mathbf{x}^*)$ and taking any p-norm on both sides, we get

$$||\mathbf{g}(\mathbf{x}) - \mathbf{g}(\mathbf{x}^*)||_p = ||\mathbf{J}_\mathbf{g}(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*) + R(\mathbf{x} - \mathbf{x}^*)||_p \tag{1.116}$$

When $\mathbf{x} \to \mathbf{x}^*$, the second and higher order terms of $\mathbf{x} - \mathbf{x}^*$ disappear and $R(\mathbf{x} - \mathbf{x}^*) \to 0$, we have

$$||\mathbf{g}(\mathbf{x}) - \mathbf{g}(\mathbf{x}^*)||_p = ||\mathbf{J}_\mathbf{g}(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*)||_p \leq ||\mathbf{J}_\mathbf{g}(\mathbf{x}^*)||_p \cdot ||\mathbf{x} - \mathbf{x}^*||_p, \tag{1.117}$$

The inequality is due to the *Cauchy-Schwarz inequality* (Section A.1) if $||\mathbf{J}_\mathbf{g}(\mathbf{x}^*)|| < 1$, the function $\mathbf{g}(\mathbf{x})$ is a contraction at $\mathbf{x}^*$.

QED

2 Unconstrained Optimization

Optimization, also known as *mathematical programming*, is a field of mathematical study with the general goal of obtaining the best solution of a given problem, and it finds a wide range of applications in many different areas, such as to minimize the error or cost, or to maximize the yields or profits. Mathematically an optimization problem can be formulated as to find the values of a set of variables represented by a vector $\mathbf{x} = [x_1, \cdots, x_N]^T$ so that an the *objective function* $f(\mathbf{x}) = f(x_1, \cdots, x_N)$ is either maximized or minimized. The solution of this problem, called the optimal solution, is denoted by $\mathbf{x}^*$. The fact that the function is either maximized or minimized is expressed as

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} f((\mathbf{x}) \qquad \text{i.e.} \qquad f(\mathbf{x}^*) \geq f(\mathbf{x}) \tag{2.1}$$

or

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f((\mathbf{x}) \qquad \text{i.e.} \qquad f(\mathbf{x}^*) \leq f(\mathbf{x}) \tag{2.2}$$

The optimization problem is *unconstrained* if the variable $\mathbf{x}$ can take any value in the domain of the function, the entire domain $\mathbb{R}^N$ of the objective function $f(\mathbf{x})$, otherwise it is *constrained* if $\mathbf{x}$ is constrained to take values only in a subset of the domain, such as $||\mathbf{x}|| < 5$.

Optimization plays an essential role in machine learning as many learning algorithms can be formulated as solving an optimization problem so that certain objective or criterion function can be either minimized or maximized. In general, we desire to minimize the difference between the observed data and the model prediction by the learning method measured in some way (e.g. the sum of squared error), or to maximizae the similarity between them measured in some way (e.g., likelihood of the model parameters). This issue will be discussed first in Section 4.1, and then through out the book chapters wherever optimization is involved in the specific algorithms.

We will consider unconstrained optimization in this chapter and constrained optimization problems in the next chapter. We will discuss various methods used for solving optimization problems in the most abstract form of maximizing or minimizing a given objective function $f(\mathbf{x})$, which will take some specific meanings in the context of various learning algorithms in the future chapters.

Two main types of algorithms widely used in unconstrained optimization are the gradient descent method and Newton's method, both of which find the opti-

mal solution iteratively based on certain approximation of the objective function. Specifically, the gradient descent method approximates the objective function by the first two terms for the constant and linear components of its Taylor expansion, while Newton's method approximates the function by the first three terms of its Taylor expandion including the third term for the quadratic component as well (see Section A.7). In other words, the gradient method depends only on the gradient vector of the function representing the local slope of the corresponding hypersurface, while Newton's method depends on the second order Hessian matrix describing more global shape of the surface as well as the local slope. Consequently, as Newton's method approximates the function more accurately, it is in general more efficient as its iteration converges more quickly than the gradient method it requires the second order derivatives which may not be availabe as conveniently as the first order derivatives. In practice, either method can be used depending on tradeoff between algorithmic efficiency and the additional cost for obtaining some extra information.

2.1 Optimization by Solving Equations

Before discussing specific methods for optimization, we first show that an unconstrained minimization problem can be converted into a problem of equation solving and visa versa, i.e., these two types of problems are equivalent in this sense.

- To minimize an objective function $f(\mathbf{x})$, we can set its gradient vector $\mathbf{g}_f(\mathbf{x})$ to zero and solve the resulting equation system:

$$\mathbf{g}_f(\mathbf{x}) = \bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = \frac{df(\mathbf{x})}{d\mathbf{x}} = \begin{bmatrix} \partial f/\partial x_1 \\ \vdots \\ \partial f/\partial x_N \end{bmatrix} = \mathbf{0} \tag{2.3}$$

  The solution $\mathbf{x}^*$ of this equation system, called a *critical, stationary, or stable point* of $f(\mathbf{x})$, is also the solution to the optimization problem, if it is not a saddle point, that either maximizes or minimizes $f(\mathbf{x})$.

  This equation system can be solved by any of the methods discussed in the previous chapter, such as the Newton-Raphson method, which finds the root of a general function $\mathbf{f}(\mathbf{x})$ iteratively: $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_f^{-1}(\mathbf{x}_n)\,\mathbf{f}(\mathbf{x}_n)$. Here, specifically, to solve the equation $\mathbf{g}_f(\mathbf{x}) = \mathbf{0}$, we first get the Jacobian $\mathbf{J}_g(\mathbf{x}) = \mathbf{H}_f(\mathbf{x})$ of the gradient $\mathbf{g}_f(\mathbf{x})$ of $f(\mathbf{x})$, which is the Hessian of $f(\mathbf{x})$, and then carry out the iteration below to eventually find $\mathbf{x}^*$ that minimizes $f(\mathbf{x})$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_g^{-1}(\mathbf{x}_n)\,\mathbf{g}_f(\mathbf{x}_n) = \mathbf{x}_n - \mathbf{H}_f^{-1}(\mathbf{x}_n)\,\mathbf{g}_f(\mathbf{x}_n) \tag{2.4}$$

  The second equality is due to the fact that the Jacobian of the gradient $\mathbf{g}_f$ of function $\mathbf{f}(\mathbf{x})$ is the Hessian of the function, i.e., $\mathbf{J}_g = \mathbf{H}_f$.

- To solve an equation system $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), \cdots, f_N(\mathbf{x})]^T = \mathbf{0}$, we can minimize an objective function defined as

$$J(\mathbf{x}) = \frac{1}{2}\mathbf{f}^T(\mathbf{x})\mathbf{f}(\mathbf{x}) = \frac{1}{2}||\mathbf{f}(\mathbf{x})||^2 = \frac{1}{2}\sum_{i=1}^{N}|f_i(\mathbf{x})|^2 \tag{2.5}$$

The solution that minimizes $J(\mathbf{x})$ is also the solution of the original equation. To see this, we again set the gradient of $J(\mathbf{x})$ zero to get as above:

$$\mathbf{g}_J(\mathbf{x}) = \bigtriangledown_{\mathbf{x}} J(\mathbf{x}) = \frac{d}{d\mathbf{x}}\left[\frac{1}{2}\mathbf{f}^T(\mathbf{x})\mathbf{f}(\mathbf{x})\right] = \mathbf{f}'(\mathbf{x})\ \ \mathbf{f}(\mathbf{x}) = \mathbf{J}_{\mathbf{f}}(\mathbf{x})\,\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{2.6}$$

As in general the Jacobian $\mathbf{J}_{\mathbf{f}}(\mathbf{x}) = \mathbf{f}'(\mathbf{x})$ is a full rank matrix, the homogeneous equation above only has a zero solution $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, which is indeed the original equation.

2.2 Newton's Method

We first consider Newton's method applied to minimizing a univariate function $f(x)$. The Taylor series expansion (Section A.7) of the function $f(x)$ around a point x_0 is:

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \cdots + \frac{1}{n!}f^{(n)}(x_0)(x - x_0)^n + \cdots \\ &\approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 = q(x) \end{aligned} \tag{2.7}$$

If $f(x)$ is a quadratic function denoted by $q(x)$, then its Taylor series contains only the first three terms (constant, linear, and quadratic terms). In this case, we can find the vertex point at which $f(x) = q(x)$ reaches its extremum, by first setting its derivative to zero:

$$\begin{aligned} q'(x) &= \frac{d}{dx}q(x) = \frac{d}{dx}\left[f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2\right] \\ &= f'(x_0) + f''(x_0)(x - x_0) = 0 \end{aligned} \tag{2.8}$$

and then solving the resulting equation to get:

$$x^* = x = x_0 - \frac{f'(x_0)}{f''(x_0)} = x_0 + \Delta x_0 \tag{2.9}$$

where $\Delta x_0 = -f'(x_0)/f''(x_0)$ is the step we need to take to move from any initial point x_0 to the solution x^* in a single step. Note that $f(x^*)$ is a minimum if $f''(x^*) > 0$ but a maximum if $f''(x^*) < 0$.

If $f(x) \neq q(x)$ is not quadratic, the result above can still be considered as an approximation of the solution, which can be improved iteratively from the initial guess x_0 to eventually approach the solution:

$$x_{n+1} = x_n + \Delta x_n = x_n - \frac{f'(x_n)}{f''(x_n)}, \qquad n = 0,\ 1,\ 2, \cdots \tag{2.10}$$

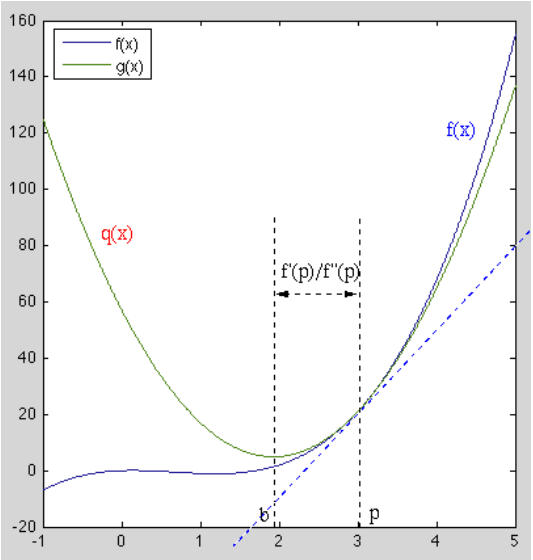


Figure 2.1 Newton's Method

where $\Delta x_n = -f'(x_n)/f''(x_n)$ is the step size.

We see that in each iterative step, the function value $f(x_n)$ is fitted by a quadratic functions $q(x_n)$ and its vertex at x_{n+1} is used as the updated approximated solution, at which the function is again fitted by another quadratic function $q(x_{n+1})$ for the next iteration, as shown in Fig. 2.1. Through this process the solution can be approached as $n \to \infty$.

We note that the iteration in Eq. (2.10) is the same as Eq. (1.47) in Chapter 1, if the function $f(x)$ is replaced by its derivative $f'(x)$. In other words, Newton's method for optimization is actually solving the equation $f'(x) = 0$, the necessary condition for an extremum of $f(x)$, by the Newton-Raphson method.

Newton's method for the minimization of a single-variable function $f(x)$ can be generalized for the minimization of a multivariable function $f(\mathbf{x}) = f(x_1, \cdots, x_N)$. Again, we approximate $f(\mathbf{x})$ by a quadratic function $q(\mathbf{x})$ containing only the first three terms of its Taylor series at some initial point $\mathbf{x}_0$:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{g}_0^T(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}_0 (\mathbf{x} - \mathbf{x}_0) = q(\mathbf{x}) \tag{2.11}$$

where $\mathbf{g}_0$ and $\mathbf{H}_0$ are respectively the gradient vector and Hessian matrix of the

function $f(\mathbf{x})$ at $\mathbf{x}_0$:

$$\mathbf{g}_0 = \mathbf{g}_f(\mathbf{x}_0) = \frac{d}{d\mathbf{x}} f(\mathbf{x}_0) = \begin{bmatrix} \frac{\partial f(\mathbf{x}_0)}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x}_0)}{\partial x_N} \end{bmatrix},$$

$$\mathbf{H}_0 = \mathbf{H}_f(\mathbf{x}_0) = \frac{d}{d\mathbf{x}}\mathbf{g}(\mathbf{x}_0) = \frac{d^2}{d\mathbf{x}^2} f(\mathbf{x}_0) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x}_0)}{\partial x_1^2} & \cdots & \frac{\partial^2 f(\mathbf{x}_0)}{\partial x_1 \partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x}_0)}{\partial x_N \partial x_1} & \cdots & \frac{\partial^2 f(\mathbf{x}_0)}{\partial x_N^2} \end{bmatrix} \quad (2.12)$$

The stationary point of $f(\mathbf{x}) = q(\mathbf{x})$ can be found from any initial guess $\mathbf{x}_0$ by setting its derivative to zero:

$$\begin{aligned} \frac{d}{d\mathbf{x}} q(\mathbf{x}) &= \frac{d}{d\mathbf{x}} \left[f(\mathbf{x}_0) + \mathbf{g}_0^T (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}_0 (\mathbf{x} - \mathbf{x}_0) \right] \\ &= \mathbf{g}_0 + \mathbf{H}_0 (\mathbf{x} - \mathbf{x}_0) = \mathbf{0} \end{aligned} \quad (2.13)$$

and solving the resulting equation to get

$$\mathbf{x}^* = \mathbf{x} = \mathbf{x}_0 - \mathbf{H}(\mathbf{x}_0)^{-1} \mathbf{g}(\mathbf{x}_0) \quad (2.14)$$

Similar to the single-variable case in which $f(x^*)$ is either a minimum or maximum depends on whether the second order derivative $f''(x^*)$ is greater or smaller than zero, here in multi-variable case the function value $f(\mathbf{x}^*)$ at the stationary point is a maximum, minimum, or saddle point, depending on the curvature of the function represented by the Hessian matrix $\mathbf{H}^* = \mathbf{H}_f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*}$ evaluated at $\mathbf{x}^*$:

- If $\mathbf{H}^* > 0$ is positive definite (all eigenvalues are positive), then $f(\mathbf{x})$ is *convex* and $f(\mathbf{x}^*)$ is the global minimum;
- If $\mathbf{H}^* < 0$ is negative definite (all eigenvalues are negative), then $f(\mathbf{x}$ is *concave* and $f(\mathbf{x}^*)$ is the global maximum;
- If $\mathbf{H}^*$ is indefinite (with both positive and negative eigenvalues), then $f(\mathbf{x})$ is neither convex nor concave, and $f(\mathbf{x}^*)$ is a saddle point (maximum in some directions, but minimum in others).

If $f(\mathbf{x})$ is quadratic and convex, the optimization problem is called *convex programming*, and its minimum can be easily obtained. (Equivalently, if $f(\mathbf{x})$ is quadratic and concave, its maximum can be easily obtained.)

But if $f(\mathbf{x}) \neq q(\mathbf{x})$ is not quadratic, the result above is not a minimum or maximum, but it can still be used as an approximation of the solution, which can be improved iteratively to approach the solution:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_n = \mathbf{x}_n - \mathbf{H}_n^{-1}\mathbf{g}_n = \mathbf{x}_n + \mathbf{d}_n, \qquad n = 0,\, 1,\, 2, \cdots \quad (2.15)$$

where $\mathbf{g}_n = \mathbf{g}(\mathbf{x}_n)$, $\mathbf{H}_n = \mathbf{H}(\mathbf{x}_n)$, and the increment $\mathbf{d}_n = \Delta\mathbf{x}_n = -\mathbf{H}_n^{-1}\mathbf{g}_n$ is called *Newton search direction*. We note that this iteration is just a generalization of $x_{n+1} = x_n - f'(x_n)/f''(x_n)$ in 1-D case where $\mathbf{g}_n = f'(x_n)$ and $\mathbf{H}_n = f''(x_n)$,

and the iteration $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}_n^{-1}\,\mathbf{g}_n$ above is just Eq. (1.68) in Chapter 1, the Newton-Raphson method applied to solving equation $f'(\mathbf{x}) = \mathbf{g}_f(\mathbf{x}) = \mathbf{0}$, as the necessary condition for an extremum of $\mathbf{f}(\mathbf{x})$. The computational complexity for each iteration is $O(N^3)$ due to the inverse operation $\mathbf{H}_n^{-1}$.

For example, we can solve an over-constrained nonlinear equation system $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ of M equations and $N < M$ unknowns by minimizing the following SSE:

$$\varepsilon(\mathbf{x}) = \frac{1}{2}||\mathbf{f}(\mathbf{x})||^2 = \frac{1}{2}\sum_{m=1}^{M} f_m^2(\mathbf{x}) \tag{2.16}$$

To use the Newton-Raphson method, we first find the gradient of the error function:

$$\mathbf{g}_\varepsilon(\mathbf{x}) = \frac{d}{d\mathbf{x}}\varepsilon(\mathbf{x}) = \frac{1}{2}\frac{d}{d\mathbf{x}}||\mathbf{f}(\mathbf{x})||^2 = \frac{d\mathbf{f}}{d\mathbf{x}}\,\mathbf{f} = \mathbf{J}_f^T\,\mathbf{f} \tag{2.17}$$

of which the ith component is:

$$g_i = \frac{\partial}{\partial x_i}\left(\frac{1}{2}||\mathbf{f}||^2\right) = \frac{1}{2}\sum_{m=1}^{M}\frac{\partial}{\partial x_i}f_m^2 = \sum_{m=1}^{M}\frac{\partial f_m}{\partial x_i}\,f_m = \sum_{m=1}^{M} J_{mi}\,f_m \tag{2.18}$$

where $J_{mi} = \partial f_m/\partial x_i$ is the component of the function's Jacobian $\mathbf{J}_f(\mathbf{x})$ in the mth row and ith column.

We further find the component of the Hessian $\mathbf{H}_\varepsilon$ of the error function in the ith row and jth column:

$$\begin{aligned} H_{ij} &= \frac{\partial^2}{\partial x_i \partial x_j}\left(\frac{1}{2}||\mathbf{f}||^2\right) = \frac{\partial}{\partial x_j}g_i = \sum_{m=1}^{M}\frac{\partial}{\partial x_j}\left[f_m\frac{\partial f_m}{\partial x_i}\right] \\ &= \sum_{m=1}^{M}\left[\frac{\partial f_m}{\partial x_i}\,\frac{\partial f_m}{\partial x_j} + f_m\frac{\partial^2 f_m}{\partial x_i \partial x_j}\right] \\ &\approx \sum_{m=1}^{M}\frac{\partial f_m}{\partial x_i}\,\frac{\partial f_m}{\partial x_j} = \sum_{m=1}^{M} J_{mi}J_{mj} \end{aligned} \tag{2.19}$$

Here J_{ij} is the component of the Jacobian $\mathbf{J}_f$ in the ith row and jth column, and we have dropped the second order terms f_m $(\partial^2 f_m/\partial x_i \partial x_j)$. Now the Hessian matrix can be written as

$$\mathbf{H}_\varepsilon(\mathbf{x}) = \frac{d^2}{d\mathbf{x}^2}\varepsilon(\mathbf{x}) = \frac{d}{d\mathbf{x}}\mathbf{g}_\varepsilon(\mathbf{x}) \approx \mathbf{J}_f^T\mathbf{J}_f \tag{2.20}$$

Based on Newton-Raphson's method, the optimal solution $\mathbf{x}^*$ that minimizes $\varepsilon(\mathbf{x})$ can be found iteratively:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}_n^{-1}\mathbf{g}_n \approx \mathbf{x}_n - (\mathbf{J}_n^T\mathbf{J}_n)^{-1}\mathbf{J}_n^T\mathbf{f}_n \tag{2.21}$$

This is the same as pseudo-inverse method in Eq. (1.77) in Chapter 1.

Example 2.1 Consider the following quadratic function for example:

$$q(\mathbf{x} = q(x_1, x_2) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} = \frac{1}{2}[x_1\ x_2]\begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{1}{2}(ax_1^2+bx_1x_2+cx_2^2)$$

The gradient vector and Hessian matrix of this function are respectively

$$\mathbf{g} = \begin{bmatrix} ax_1 + bx_2/2 \\ bx_1/2 + cx_2 \end{bmatrix}, \qquad \mathbf{H} = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix} = \mathbf{A}$$

The two eigenvalues of the symmetric matrix $\mathbf{H}$ can be found to be

$$\lambda_{1,2} = \frac{1}{2}[(a+c) \pm \sqrt{(a+c)^2 - 4ac + b^2}] \tag{2.22}$$

and the determinant of $\mathbf{H}$ is $\det \mathbf{H} = \lambda_1\lambda_2 = ac - b^2/4$. The shape and extreme of the surface represented by $q(\mathbf{x})$ are determined by the coefficients let a, b, c, which can take any values such as those discussed below and shown in Fig. 2.2.

- $a = 3,\ b = 2,\ c = 4,\ \lambda_1 = 2.382,\ \lambda_2 = 4.618$ are both positive, and $\mathbf{H}$ is positive definite. $q(\mathbf{x})$ has a minimum at $q(0,\ 0) = 0$.
- $a = -3,\ b = 2,\ c = -4,\ \lambda_1 = -4.618\ \lambda_2 = -2.382$ are both negative, and $\mathbf{H}$ is negative definite. $q(\mathbf{x})$ has a maximum at $q(0,\ 0) = 0$.
- $a = 3,\ b = \sqrt{4ac},\ c = 4,\ \lambda_1 = 7,\ \lambda_2 = 0$, and $\mathbf{H}$ is positive semi-definite. $q(0, 0)$ is a mininum along one direction but constant along another.
- $a = -3,\ b = 2,\ c = 4,\ \lambda_1 = -3.140,\ \lambda_2 = 4.140$ are of different signs, and $\mathbf{H}$ is indefinite. $q(0,\ 0) = 0$ is a saddle point, which is a maximum along one direction but a minimum along another.

These properties based on the (semi) definiteness of the Hessian $\mathbf{H}$ can be generalized to quadratic functions with more variables than two (see Section A.2.3).

We can speed up the convergence by a bigger step size $\delta > 1$. However, if δ is too big, the solution may be skipped and the iteration may not converge if it gets into an oscillation around the solution. Even worse, the iteration may become divergent. For such reasons, a smaller step size $\delta < 1$ may be preferred sometimes.

In summary, Newton's method approximates the function $f(\mathbf{x})$ at an estimated solution $\mathbf{x}_n$ by a quadratic equation (the first three terms of the Taylor's series) based on the gradient $\mathbf{g}_n$ and Hessian $\mathbf{H}$ of the function at $\mathbf{x}_n$, and treat the vertex of the quadratic equation as the updated estimate $\mathbf{x}_{n+1}$.

Newton's method requires the Hessian matrix as well as the gradient to be available. Moreover, it is necessary calculate the inverse of the Hessian matrix in each iteration, which may be computationally expensive.

Example 2.2 The Newton's method is applied to solving the following non-

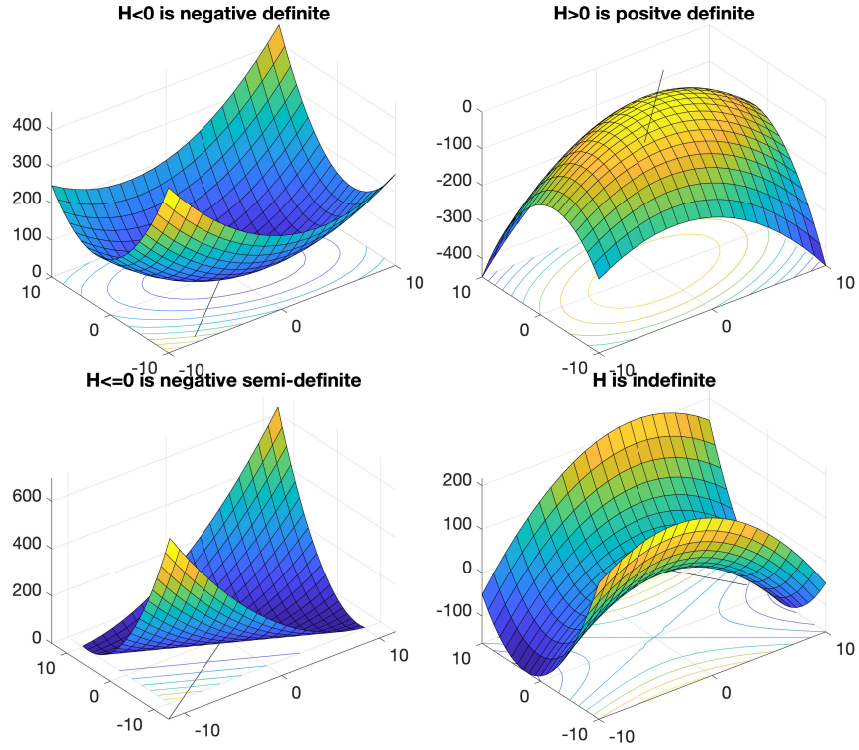


Figure 2.2 Quadratic Surfaces
The contour lines of the quadratic function are shown in color coded lines and the gradient vectors at the extreme/saddle points are shown in black.

linear equation system of $N = 3$ variables:

$$\begin{cases} f_1(x_1,\, x_2,\, x_3) = 3x_1 - (x_2 x_3)^2 - 3/2 \\ f_2(x_1,\, x_2,\, x_3) = 4x_1^2 - 625\, x_2^2 + 2x_2 - 1 \\ f_3(x_1,\, x_2,\, x_3) = exp(-x_1 x_2) + 20x_3 + 9 \end{cases}$$

with the exact solution $(x_1 = 0.5,\ x_2 = 0,\ x_3 = -0.5)$. These equations can be expressed in vector form as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ and solved as an optimization problem with the objective function $J(\mathbf{x}) = \mathbf{f}^T(\mathbf{x})\mathbf{f}(\mathbf{x})$. The iteration from an initial guess

$\mathbf{x}_0 = \mathbf{0}$ is shown below.

n	$\mathbf{x} = [x_1, x_2, x_3]$	$\|\|\mathbf{f}(\mathbf{x})\|\|$
0	0.000000, 0.000000, 0.000000	$1.016120e+01$
1	0.500000, 0.500000, −0.500000	$1.552502e+02$
2	0.499550, 0.250800, −0.493801	$3.881300e+01$
3	0.500096, 0.126206, −0.496852	$9.702208e+00$
4	0.500025, 0.063914, −0.498405	$2.425198e+00$
5	0.500010, 0.032778, −0.499181	$6.059054e-01$
6	0.500005, 0.017231, −0.499570	$1.510777e-01$
7	0.500003, 0.009498, −0.499763	$3.737330e-02$
8	0.500002, 0.005712, −0.499857	$8.959365e-03$
9	0.500001, 0.003968, −0.499901	$1.900145e-03$
10	0.500001, 0.003326, −0.499917	$2.577603e-04$
11	0.500001, 0.003206, −0.499920	$8.932714e-06$
12	0.500001, 0.003202, −0.499920	$1.238536e-08$
13	0.500001, 0.003202, −0.499920	$2.371437e-14$

(2.23)

We see that after 13 iterations the algorithm converges to the following approximated solution with accuracy of $J(\mathbf{x}) = ||\mathbf{f}(\mathbf{x}^*)||^2 \approx 10^{-28}$:

$$\mathbf{x}^* = \begin{bmatrix} 0.5000008539707297 \\ 0.0032017070323056 \\ -0.4999200212218281 \end{bmatrix} \tag{2.24}$$

2.3 Gradient Descent Method

Newton's method discussed above is based on the gradient $\mathbf{g}_f(\mathbf{x})$ and Hessian $\mathbf{H}_f(\mathbf{x})$, representing the local variation and more global curvature of the function $f(\mathbf{x})$ to be minimized. The method is not applicable if the Hessian $\mathbf{H}_f$ is not available, because either the analytical form of the function is not known or the cost for computing the inverse $\mathbf{H}_f^{-1}$ is too high. In such cases, we can use the gradient descent method based only on the gradient but no longer the Hessian.

We first consider the minimization of a single-variable function $f(x)$. From any initial point x_0, we can move to a nearby point x along the opposite direction of the gradient, so that the function value will be smaller than that at the current position x_0:

$$x = x_0 - \delta f'(x_0) = x_0 + \Delta x_0, \quad \text{where} \quad \Delta x_0 = -\delta f'(x_0), \quad \delta > 0 \tag{2.25}$$

We see that no matter whether $f'(x_0)$ is positive or negative, the function value $f(x)$ approximated by the first two terms of its Taylor series is always reduced if the positive step size *delta* is small enough:

$$f(x) \approx f(x_0) + f'(x_0)\Delta x_0 = f(x_0) - |f'(x_0)|^2\delta < f(x_0) \tag{2.26}$$

This process can be carried out iteratively

$$x_{n+1} = x_n - \delta_n \, f'(x_n), \qquad n = 0, \, 1, \, 2, \cdots \tag{2.27}$$

until eventually reaching a point x^* at which $f'(x^*) = 0$ and no further progress can be made, i.e. a local minimum of the function is reached.

This simple method can be generalized to minimize a multi-variable objective function $f(\mathbf{x}) = f(x_1, \cdots, x_N)$ in N-D space. The derivative of the 1-D case is generalized to the gradient vector $\mathbf{g}(\mathbf{x}) = df(\mathbf{x})/d\mathbf{x}$ of function $f(\mathbf{x})$, which is in the direction along which the function increases most rapidly with the steepest slope, perpendicular to the contour or iso-lines of the function $f(\mathbf{x})$. The fastest way to reduce $f(\mathbf{x})$ is to go down hill along the opposite direction of the gradient vector.

Specifically the gradient descent method (also called steepest descent or down hill method) carries out the iteration $\mathbf{x} = \mathbf{x}_0 + \Delta\mathbf{x}$, which is can be considered as the first two terms of the Taylor series of $f(\mathbf{x})$ with $\Delta\mathbf{x} = -\delta\mathbf{g}$ ($\delta > 0$):

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{g}_0^T \Delta\mathbf{x} = f(\mathbf{x}_0) - \delta \mathbf{g}_0^T \mathbf{g}_0 = f(\mathbf{x}_0) - \delta ||\mathbf{g}||^2 < f(\mathbf{x}_0) \tag{2.28}$$

As the function value is always reduced by this iteration, it can be carried out repeatedly

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \delta_n \, \mathbf{g}_n = (\mathbf{x}_{n-1} - \delta_{n-1} \, \mathbf{g}_{n-1}) - \delta_n \, \mathbf{g}_n = \cdots = \mathbf{x}_0 - \sum_{i=0}^{n} \delta_n \, \mathbf{g}_i \tag{2.29}$$

until eventually reaching a point $\mathbf{x}^*$ at which $\mathbf{g}(\mathbf{x}^*) = \mathbf{0}$ and $f(\mathbf{x})$ is minimized.

Comparing Eq. (2.29) with Newton's method where the search direction is $\mathbf{d}_n = -\mathbf{H}_n^{-1}\mathbf{g}_n$ in Eq. (2.15) based on both $\mathbf{H}_n$ and $\mathbf{g}_n$, we see that here in gradient descent method the search direction $\mathbf{d}_n = -\mathbf{g}_n$ relies only on the local information contained in the gradient $\mathbf{g}$ without more global information contained in the second order Hessian $\mathbf{H}$, i.e., the gradient method does not have as much information as Newton's method and therefore may not be as efficient. For example, when the function is quadratic, as discussed before, Newton's method can find the solution in a single step from any initial guess, but it takes the gradient descent method multiple steps to reach the solution, as it always follows the negative direction of the local gradient, which does not lead toward the solution directly in general. However, the gradient descent method is computationally less expensive as the Hessian matrix is not needed.

Example 2.3 Consider a two-variable quadratic function:

$$\begin{aligned} f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} &= [x_1 \; x_2] \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= [x_1 \; x_2] \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 2x_1^2 + 2x_1x_2 + x_2^2 \end{aligned}$$

where **A**is a symmetric positive definite matrix. This function has a minimum $f(x_1, \, x_2) = 0$ at $x_1 = x_2 = 0$.

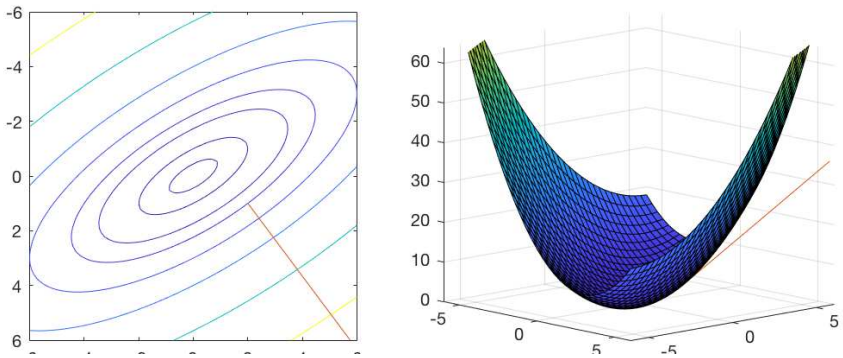


Figure 2.3 Gradient of a Quadratic Surface

We select an initial guess $\mathbf{x}_0 = [1,\ 2]^T$, at which the gradient is $\mathbf{g}_0 = [4,\ 3]^T$, as shown in Fig. 2.3. Now we compare the gradient method with Newton's method, in terms of the search direction $\mathbf{d}$ and progress of the iterations:

- Newton's method: Here we have

$$\mathbf{g} = \begin{bmatrix} 2x_1 + x_2 \\ x_1 + x_2 \end{bmatrix}, \quad \mathbf{H} = \mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{H}^{-1} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}$$

  and the search direction is $\mathbf{d}_0 = -\mathbf{H}^{-1}\mathbf{g}_0 = -[1,\ 2]^T$ (the red arrow in the figure). In a single iteration we reach

$$\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{H}^{-1}\mathbf{g}_0 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

  which is the minimum of the function.
- The gradient descent method: the search direction is $\mathbf{d}_0 = -\mathbf{g}_0 = -[4,\ 3]^T$, perpendicular to the contour of the function. The first iteration is:

$$\mathbf{x}_1 = \mathbf{x}_0 - \delta\mathbf{g}_0 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \delta \begin{bmatrix} 4 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 - \delta 4 \\ 2 - \delta 3 \end{bmatrix}$$

  We need to determine the step size δ (to be considered later) to find $\mathbf{x}_1$, and then continue the iteration.

Fig. 2.4 compares the different search directions by Newton's method (red) and the gradient descent method (blue). We see that the solution is reached in a single step in Newton's method, based on the Hessian matrix $\mathbf{H}$ representing some global information of the elliptical shape of the contour line of the quadratic function, as well as the gradient $\mathbf{g}_0$ representing the local information at $\mathbf{x}_0$. In comparison, the search direction of the gradient descent method is based on only the local information in the gradient $\mathbf{g}_0$ without more global information in $\mathbf{H}$, consequently multiple steps are needed for the iterativon to approach the

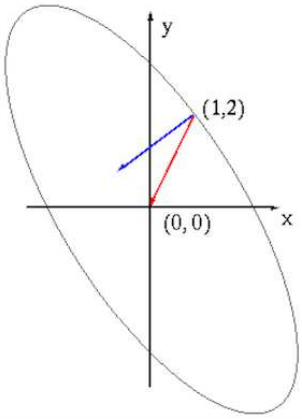


Figure 2.4 Comparison of Gradient Descent and Newton's Method

solution. The gradient descent method is less effective than Newton's method due to the lack of second order derivative carrying more global information.

In the gradient descent method, we also need to determine a proper step size δ. If δ is too small, the iteration may converge too slowly, especially when $f(x)$ reduces slowly toward its minimum. On the other hand, if δ is too large but $f(x)$ has some rapid variations in the local region, the minimum of the function may be skipped and the iteration may not converge. We will consider how to find the optimal step size in the next section.

2.4 Line minimization

In general, in any iterative algorithm for minimizing a function $f(\mathbf{x})$, including Newton's method (Eq. (2.15)) and the gradient descent method (Eq. (2.29)), the variable $\mathbf{x}$ is updated iteratively $\mathbf{x}_{n+1} = \mathbf{x}_n + \delta_n \mathbf{d}_n$ based on both the search direction $\mathbf{d}_n$ and the step size δ, which need to be determined in such a way that the function value $f(\mathbf{x})$ is maximally reduced at each iteration.

First, the search direction $\mathbf{d}_n$ needs to point away from the gradient $\mathbf{g}_n$ along which $f(\mathbf{x}_n)$ increases most rapidly. In other words, the angle between $\mathbf{d}_n$ and $\mathbf{g}_n$ should be greater than $\pi/2$:

$$\cos^{-1}\left(\frac{\mathbf{d}_n^T \mathbf{g}_n}{||\mathbf{d}_n||\,||\mathbf{g}_n||}\right) > \frac{\pi}{2} \qquad i.e., \qquad \mathbf{d}_n^T \mathbf{g}_n < 0 \tag{2.30}$$

This condition is indeed satisfied in both the gradient descent method with $\mathbf{d}_n = -\mathbf{g}_n$ and Newton's method with $\mathbf{d}_n = -\mathbf{H}_n^{-1}\mathbf{g}_n$:

$$\mathbf{d}_n^T \mathbf{g}_n = -\mathbf{g}_n^T \mathbf{H}_n^{-1} \mathbf{g}_n < 0, \qquad \mathbf{d}_n^T \mathbf{g}_n = -\mathbf{g}_n^T \mathbf{g}_n = -||\mathbf{g}_n||^2 < 0 \tag{2.31}$$

Here $\mathbf{H}_n$ is assumed to be positive definite so that a minimum exists.

Next, the optimal step size δ_n needs to be such that the function value $f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \delta_n \mathbf{d}_n)$ is minimized along the search direction $\mathbf{d}_n$. To do so, we set to zero the derivative of the function value with respect to δ_n, and get the *directional derivative* (Section A.4.3) along the direction of $\mathbf{d}_n$ based on the chain rule:

$$\frac{d}{d\delta_n} f(\mathbf{x}_{n+1}) = \frac{d}{d\delta_n} f(\mathbf{x}_n + \delta_n \mathbf{d}_n) = \left(\frac{d\, f(\mathbf{x}_{n+1})}{d\mathbf{x}} \right)^T \frac{d(\mathbf{x}_n + \delta_n \mathbf{d}_n)}{d\delta_n} = \mathbf{g}_{n+1}^T \mathbf{d}_n = 0 \tag{2.32}$$

This result indicates that ideally the gradient $\mathbf{g}_{n+1} = f'(\mathbf{x}_{n+1})$ at the next point $\mathbf{x}_{n+1}$ should be perpendicular to the search direction $\mathbf{d}_n$. In other words, when traversing along $\mathbf{d}_n$, we should stop at the point $\mathbf{x}_{n+1} = \mathbf{x}_n + \delta \mathbf{d}_n$ at which the gradient $\mathbf{g}_{n+1}$ is perpendicular to $\mathbf{d}_n$, i.e., it has zero component along $\mathbf{d}_n$, and the corresponding δ is the optimal step size.

To find such an optimal step size δ, we can consider $f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \delta \mathbf{d}_n)$ as a function of δ treated as the single variable, and find the its optimal value that minimizes $f(\mathbf{x}_{n+1})$. To do so, we assume the value of δ is small enough so that this function can be pproximated by the first three terms of its Taylor series at $\delta = 0$ (Maclaurin series of function $f(\delta)$):

$$\begin{aligned} & f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \delta \mathbf{d}_n) \\ & \approx [f(\mathbf{x}_n + \delta \mathbf{d}_n)]_{\delta=0} + \delta \left[\frac{d}{d\delta} f(\mathbf{x}_n + \delta \mathbf{d}_n) \right]_{\delta=0} + \frac{\delta^2}{2} \left[\frac{d^2}{d\delta^2} f(\mathbf{x}_n + \delta \mathbf{d}_n) \right]_{\delta=0} \end{aligned} \tag{2.33}$$

The three terms of this expression can be further written as

$$[f(\mathbf{x}_n + \delta \mathbf{d}_n)]_{\delta=0} = f(\mathbf{x}_n) \tag{2.34}$$

$$\left[\frac{d}{d\delta} f(\mathbf{x}_n + \delta \mathbf{d}_n) \right]_{\delta=0} = \left[\mathbf{g}(\mathbf{x}_n + \delta \mathbf{d}_n)^T \mathbf{d}_n \right]_{\delta=0} = \mathbf{g}_n^T \mathbf{d}_n \tag{2.35}$$

$$\begin{aligned} \left[\frac{d^2}{d\delta^2} f(\mathbf{x}_n + \delta \mathbf{d}_n) \right]_{\delta=0} &= \left[\frac{d}{d\delta} \mathbf{g}(\mathbf{x}_n + \delta \mathbf{d}_n)^T \right]_{\delta=0} \frac{d}{d\delta} (\mathbf{x}_n + \delta \mathbf{d}_n) \\ &= \left[\frac{d}{d\mathbf{x}} \mathbf{g}(\mathbf{x}) \; \frac{d}{d\delta} (\mathbf{x}_n + \delta \mathbf{d}_n) \right]_{\delta=0}^T \mathbf{d}_n \\ &= (\mathbf{H}_n \mathbf{d}_n)^T \mathbf{d}_n = \mathbf{d}_n^T \mathbf{H}_n \mathbf{d}_n \end{aligned} \tag{2.36}$$

where $\mathbf{H}_n = \mathbf{H}_n^T$ is the Hessian matrix of $f(\mathbf{x})$ at $\mathbf{x}_n$. Substituting these back into Eq. (2.33) we get:

$$f(\mathbf{x}_n + \delta \mathbf{d}_n) \approx f(\mathbf{x}_n) + \delta\, \mathbf{g}_n^T\, \mathbf{d}_n + \frac{\delta^2}{2}\, \mathbf{d}_n^T \mathbf{H}_n\, \mathbf{d}_n \tag{2.37}$$

To find the optimal δ that minimizes $f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \delta \mathbf{d}_n)$, we first find its derivative with respect to δ and then set it to zero:

$$\begin{aligned} \frac{d}{d\delta} f(\mathbf{x}_n + \delta \mathbf{d}_n) &\approx \frac{d}{d\delta} \left(f(\mathbf{x}_n) + \delta \mathbf{g}_n^T\, \mathbf{d}_n + \frac{\delta^2}{2} \mathbf{d}_n^T \mathbf{H}_n\, \mathbf{d}_n \right) \\ &= \mathbf{g}_n^T\, \mathbf{d}_n + \delta\, \mathbf{d}_n^T \mathbf{H}_n\, \mathbf{d}_n = 0 \end{aligned} \tag{2.38}$$

Solving this equation for δ we get:

$$\delta_n = -\frac{\mathbf{g}_n^T \mathbf{d}_n}{\mathbf{d}_n^T \mathbf{H}_n \mathbf{d}_n} \tag{2.39}$$

Note that only if the value of this δ_n is small enough for the approximation in Eq. (2.33) to be good enough, will it be a good approximation of the true optimal step size (and each search directions is perpendicular to the previous one). Otherwise it may not be be very different from the true optimal step size.

Based on this result, we can get the optimal step side for both the Newton's method and gradient descent method as shown below:

- **Newton's method:**

  The search direction is $\mathbf{d}_n = -\mathbf{H}_n^{-1}\mathbf{g}_n$, and the optimal step size is

$$\delta_n = -\frac{\mathbf{g}_n^T \mathbf{d}_n}{\mathbf{d}_n^T \mathbf{H}_n \mathbf{d}_n} = \frac{\mathbf{g}_n^T (\mathbf{H}_n^{-1}\mathbf{g}_n)}{(\mathbf{H}_n^{-1}\mathbf{g}_n)^T \mathbf{H}_n (\mathbf{H}_n^{-1}\mathbf{g}_n)} = \frac{\mathbf{g}_n^T (\mathbf{H}_n^{-1}\mathbf{g}_n)}{(\mathbf{H}_n^{-1}\mathbf{g}_n)^T \mathbf{g}_n} = 1 \tag{2.40}$$

  This confirms that the iteration in Eq. (2.15) is indeed optimal:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \delta_n \mathbf{d}_n = x_n - \delta_n \mathbf{H}_n^{-1}\mathbf{g}_n = \mathbf{x}_n - \mathbf{H}_n^{-1}\mathbf{g}_n \tag{2.41}$$

- **Gradient descent method:**

  The search direction is $\mathbf{d}_n = -\mathbf{g}_n$, and Eq. (2.32) becomes:

$$\mathbf{g}_{n+1}^T \mathbf{d}_n = -\mathbf{g}_{n+1}^T \mathbf{g}_n = 0, \quad \text{i.e.,} \quad \mathbf{g}_{n+1} \perp \mathbf{g}_n \tag{2.42}$$

  i.e., the search direction $\mathbf{d}_{n+1} = -\mathbf{g}_{n+1}$ is always perpendicular to the previous one $\mathbf{d}_n = -\mathbf{g}_n$, i.e., the iteration follows a zigzag path composed of a sequence of segments from the initial guess to the final solution. The optimal step size is

$$\delta_n = -\frac{\mathbf{g}_n^T \mathbf{d}_n}{\mathbf{d}_n^T \mathbf{H}_n \mathbf{d}_n} = \frac{\mathbf{g}_n^T \mathbf{g}_n}{\mathbf{g}_n^T \mathbf{H}_n \mathbf{g}_n} = \frac{||\mathbf{g}_n||^2}{\mathbf{g}_n^T \mathbf{H}_n \mathbf{g}_n} \tag{2.43}$$

Given $\mathbf{H}_n$ as well as $\mathbf{g}_n$, we can find the optimal step size δ_n with complexity $O(N^2)$ for each iteration. However, as we assume the Hessian $\mathbf{H}_n$ is not available in the gradient descent method, the optimal step size above cannot be computed. We can instead approximate $f''(\delta)\big|_{\delta=0}$ in the third term of Eq. (2.33) at two nearby points at $\delta = 0$ and $\delta = \sigma$, where σ is a small value:

$$\begin{aligned}\left[\frac{d^2}{d\delta^2} f(\mathbf{x}+\delta\mathbf{d})\right]_{\delta=0} &= \left[\frac{d}{d\delta} f'(\mathbf{x}+\delta\mathbf{d})\right]_{\delta=0} = \lim_{\sigma\to 0} \frac{f'(\mathbf{x}+\sigma\mathbf{d}) - f'(\mathbf{x})}{\sigma} \\ &\approx \frac{\mathbf{g}^T(\mathbf{x}+\sigma\mathbf{d})\,\mathbf{d} - \mathbf{g}^T(\mathbf{x})\,\mathbf{d}}{\sigma} = \frac{\mathbf{g}_\sigma^T \mathbf{d} - \mathbf{g}^T \mathbf{d}}{\sigma}\end{aligned} \tag{2.44}$$

where $\mathbf{g} = \mathbf{g}(\mathbf{x})$ and $\mathbf{g}_\sigma = \mathbf{g}(\mathbf{x}+\sigma\mathbf{d})$. This approximation can be used to replace $\mathbf{d}_n^T \mathbf{H}_n \mathbf{d}_n$ in Eq. (2.38) above:

$$\frac{d}{d\delta} f(\mathbf{x}_n + \delta\mathbf{d}_n) \approx \mathbf{g}_n^T\,\mathbf{d}_n + \frac{\delta}{\sigma}\left(\mathbf{g}_{\sigma n}^T \mathbf{d}_n - \mathbf{g}_n^T \mathbf{d}_n\right) = 0 \tag{2.45}$$

Solving for δ we get the estimated optimal step size:

$$\delta_n = -\frac{\sigma\, \mathbf{g}_n^T \mathbf{d}_n}{\mathbf{g}_{\sigma n}^T \mathbf{d}_n - \mathbf{g}_n^T \mathbf{d}_n} = -\frac{\sigma\, \mathbf{g}_n^T \mathbf{d}_n}{(\mathbf{g}_{\sigma n} - \mathbf{g}_n)^T \mathbf{d}_n} \tag{2.46}$$

Specifically in the gradient descent method with $\mathbf{d}_n = -\mathbf{g}_n$, this optimal step size becomes:

$$\delta_n = -\frac{\sigma\, \mathbf{g}_n^T \mathbf{d}_n}{(\mathbf{g}_{\sigma n} - \mathbf{g}_n)^T \mathbf{d}_n} = -\frac{\sigma\, \mathbf{g}_n^T \mathbf{g}_n}{(\mathbf{g}_{\sigma n} - \mathbf{g}_n)^T \mathbf{g}_n} = \frac{\sigma\, ||\mathbf{g}_n||^2}{||\mathbf{g}_n||^2 - \mathbf{g}_{\sigma n}^T \mathbf{g}_n} \tag{2.47}$$

and the iteration becomes

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \delta_n \mathbf{g}_n = \mathbf{x}_n + \frac{\sigma\, ||\mathbf{g}_n||^2}{\mathbf{g}_{\sigma n}^T \mathbf{g}_n - ||\mathbf{g}_n||^2}\, \mathbf{g}_n \tag{2.48}$$

Example 2.4 The gradient descent method applied to solve the same three-variable equation system previously solved by Newton's method:

$$\begin{cases} f_1(x_1,\, x_2,\, x_3) = 3x_1 - (x_2 x_3)^2 - 3/2 \\ f_2(x_1,\, x_2,\, x_3) = 4x_1^2 - 625\, x_2^2 + 2x_2 - 1 \\ f_3(x_1,\, x_2,\, x_3) = exp(-x_1 x_2) + 20x_3 + 9 \end{cases}$$

The step size δ_n is determined by the secant method with $\sigma = 10^{-6}$. The iteration from an initial guess $\mathbf{x}_0 = \mathbf{0}$ is shown below:

n	$\mathbf{x} = [x_1,\, x_2,\, x_3]$	$\|\|\mathbf{f}(\mathbf{x})\|\|$
0	0.0000, 0.0000, 0.0000	$1.032500e+02$
10	0.4246, −0.0073, −0.5002	$1.535939e-01$
20	0.5015, 0.0064, −0.4998	$2.448241e-05$
30	0.5009, 0.0057, −0.4998	$9.178209e-06$
40	0.5006, 0.0052, −0.4998	$4.17587e-06$
50	0.5004, 0.0049, −0.4999	$2.122594e-06$
60	0.5003, 0.0047, −0.4999	$1.182466e-06$
100	0.5001, 0.0043, −0.4999	$1.805871e-07$
150	0.5000, 0.0041, −0.4999	$2.720744e-08$
200	0.5000, 0.0041, −0.4999	$4.934027e-09$
250	0.5000, 0.0040, −0.4999	$9.630085e-10$
300	0.5000, 0.0040, −0.4999	$1.939747e-10$
350	0.5000, 0.0040, −0.4999	$3.961646e-11$
400	0.5000, 0.0040, −0.4999	$8.140393e-12$
450	0.5000, 0.0040, −0.4999	$1.677968e-12$
500	0.5000, 0.0040, −0.4999	$3.462695e-13$
550	0.5000, 0.0040, −0.4999	$7.136628e-14$
600	0.5000, 0.0040, −0.4999	$1.474205e-14$

We see that after the first 100 iterations the error is reduced to about $J(\mathbf{x}) = ||\mathbf{f}(\mathbf{x}^*)||^2 \approx 10^{-14}$, and With 500 additional iterations the algorithm converges

to the following approximated solution with accuracy of $J(\mathbf{x}) \approx 10^{-28}$:

$$\mathbf{x}^* = \begin{bmatrix} 0.5000013623816102 \\ 0.0040027495837189 \\ -0.4999000311539049 \end{bmatrix} \tag{2.49}$$

Although the gradient descent method requires many more iterations than Newton's method to converge, the computational cost in each iteration is much reduced, as no more matrix inversion is needed.

When it is difficult or too computationally costly to find the optimal step size along the search direction, some suboptimal step size may be acceptable, such as in the quasi-Newton methods (Section 2.5) for minimization problems. In this case, although the step size δ is no longer required to be such that the function value at the new position $f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \delta\mathbf{d}_n)$ is minimized along the search direction $\mathbf{d}_n$, the step size δ still has to satisfy the following *Wolfe conditions*:

- *Sufficient decrease (Armijo rule):*

$$f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \delta\mathbf{d}_n) \leq f(\mathbf{x}_n) + c_1\,\delta\mathbf{d}^T\mathbf{g}_n \tag{2.50}$$

- *Curvature condition:*

$$\mathbf{g}_{n+1}^T\mathbf{d}_n \geq c_2\,\mathbf{g}_n^T\mathbf{d}_n \tag{2.51}$$

  As $\mathbf{g}_n^T\mathbf{d}_n < 0$, this condition can also be written in the following alternative form:

$$|\mathbf{g}_{n+1}^T\mathbf{d}_n| < |c_2\,\mathbf{g}_n^T\mathbf{d}_n| \tag{2.52}$$

Here the two constants c_1 and c_2 above satisfy $0 < c_1 < c_2 < 1$.

In general, these conditions are motivated by the desired effect that after each iterative step, the function should have a shallower slope along $\mathbf{d}_n$, as well as a lower value, so that eventually the solution can be approached where $f(\mathbf{x})$ is minimum and the gradient is zero.

Specifically, to understand the first condition above, we represent the function to be minimized as a single-variable function of the step size $\phi(\delta) = f(\mathbf{x}_n + \delta\mathbf{d}_n)$, and its tangent line at the point $\delta = 0$ as a linear function $L_0(\delta) = a + b\delta$, where the intercept a can be found as $a = L_0(0) = \phi(\delta)\big|_{\delta=0} = f(\mathbf{x}_n)$, and the slope b can be found as the derivative of $\phi(\delta) = f(\mathbf{x}_n + \delta\mathbf{d}_n)$ at $\delta = 0$:

$$b = \frac{d}{d\delta}\, f(\mathbf{x}_n + \delta\mathbf{d}_n)\bigg|_{\delta=0} = \mathbf{g}_n^T\mathbf{d}_n < 0 \tag{2.53}$$

which is required to be negative for the function value to be reduced, $f(\mathbf{x}_n + \delta\mathbf{d}_n) < f(\mathbf{x}_n)$. Now the function of the tangent line can be written as

$$L_0(\delta) = a + b\delta = f(\mathbf{x}_n) + \mathbf{g}_n^T\mathbf{d}_n\delta \tag{2.54}$$

Comparing this with a constant line $L_1(\delta) = f(\mathbf{x}_n)$ of slope zero, we see that any straight line between $L_0(\delta)$ and $L_1(\delta)$ can be described by $L(\delta) = f(\mathbf{x}_n) +$

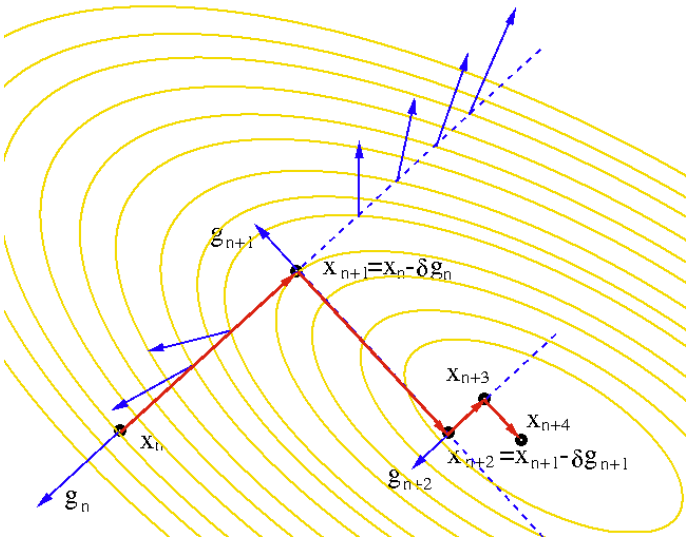


Figure 2.5 Iterations of the Gradient Descent Method

$c_1\,\mathbf{g}_n^T\mathbf{d}_n\delta$ with $0 < c_1 < 1$, with a slope $0 < c_1\mathbf{g}_n^T\mathbf{d}_n < \mathbf{g}_n^T\mathbf{d}_n$. The Armijo rule is to find any $\delta > 0$ that satisfies

$$f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \delta\mathbf{d}_n) \leq f(\mathbf{x}_n) + c_1\,\mathbf{g}_n^T\mathbf{d}_n\delta < f(\mathbf{x}_n) \tag{2.55}$$

We see that the value of $f(\mathbf{x})$ is guaranteed to be reduced.

The second condition requires that at the new position $\mathbf{x}_{n+1}$ the slope of the gradient $\mathbf{g}_{n+1}$ along the search direction $\mathbf{d}_n$ be sufficiently reduced to be less than a specified value (determined by c_2), in comparison to the slope at the old position $\mathbf{x}_n$.

The reason why $\mathbf{g}_{n+1} \perp \mathbf{g}_n$ can also be explained geometrically. As shown in Fig. 2.5, the gradient vectors at various points along the direction of $-\mathbf{g}_n$ are shown by the blue arrows, and their projections onto the direction represent the slopes of the function $f(\delta_n) = f(\mathbf{x}_n - \delta_n\mathbf{g}_n)$. Obviously at $\mathbf{x}_{n+1}$ where $f(\delta_n)$ reaches its minimum, its slope is zero. In other words, the projection of the gradient $\mathbf{g}_{n+1}$ onto the direction of $-\mathbf{g}_n$ is zero, i.e., $\mathbf{g}_{n+1} \perp \mathbf{g}_n$ or $\mathbf{g}_{n+1}^T\mathbf{g}_n = 0$.

The gradient descent method gradually approaches a solution $\mathbf{x}$ of an N-D minimization problem by moving from the initial guess $\mathbf{x}_0$ along a zigzag path composed of a set of segments with any two consecutive segments perpendicular to each other. The number of steps depends greatly on the initial guess. As illustrated in Fig. 2.6 for a quadratic surface over a 2-D space, the best possible case (blue) is that the solution happens to be on the gradient direction of the initial guess, which could be reached in a single step, while the worst possible case (red) is that the gradient direction of the initial guess happens to be 45 degrees off from the gradient direction of the optimal case, and it takes many

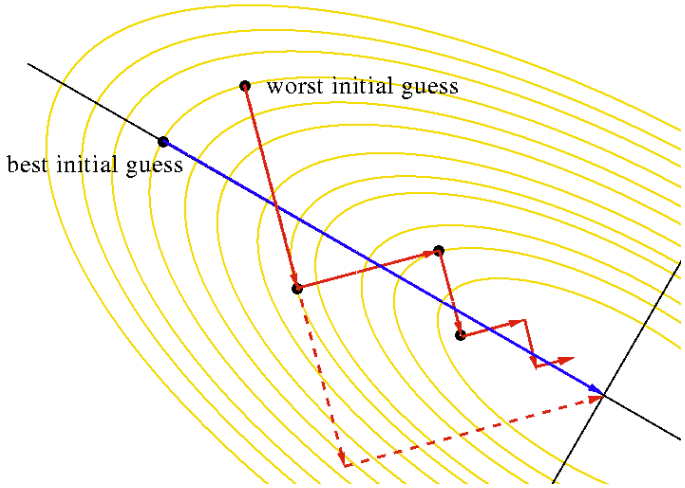


Figure 2.6 Best (blue) and Worst (red) Initial Guess

zigzag steps to go around the optimal path to reach the solution. Many of the steps are in the same direction as some of the previous steps.

To improve the performance of the gradient descent method we can include in the iteration a momentum term representing the search direction previously traversed:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \delta_n \mathbf{g}_n + \alpha_n(\mathbf{x}_n - \mathbf{x}_{n-1}) \tag{2.56}$$

Now two consecutive search directions are no longer perpendicular to each other and the resulting search path is smoother than the zigzag path without the momentum term. The parameter α_m controls how much momentum is to be added.

Obviously it is most desirable not to repeat any of the previous directions traveled so that the solution can be reached in N steps, each in a unique direction in the N-D space. In other words, the subsequent steps are independent of each other, never interfering with the results achieved in the previous steps. Such a method will be discussed in the next section.

Example 2.5 The Rosenbrock function

$$f(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2 \qquad \text{(e.g., } a = 1,\ b = 100\text{)}$$

is a two-variable non-convex function with a global minimum $f(x_1,\ x_2) = 0$ at the point $(1,\ 1)$, which is inside a long parabolic shaped valley as shown in the left panel of Fig. 2.7. As the slope along the valley is very shallow, it is difficult for an algorithm to converge quickly to the minimum. For this reason, the Rosenbrock function is often used to test various minimization algorithms.

Bothe the gradient method and Newton's method are used to find the min-

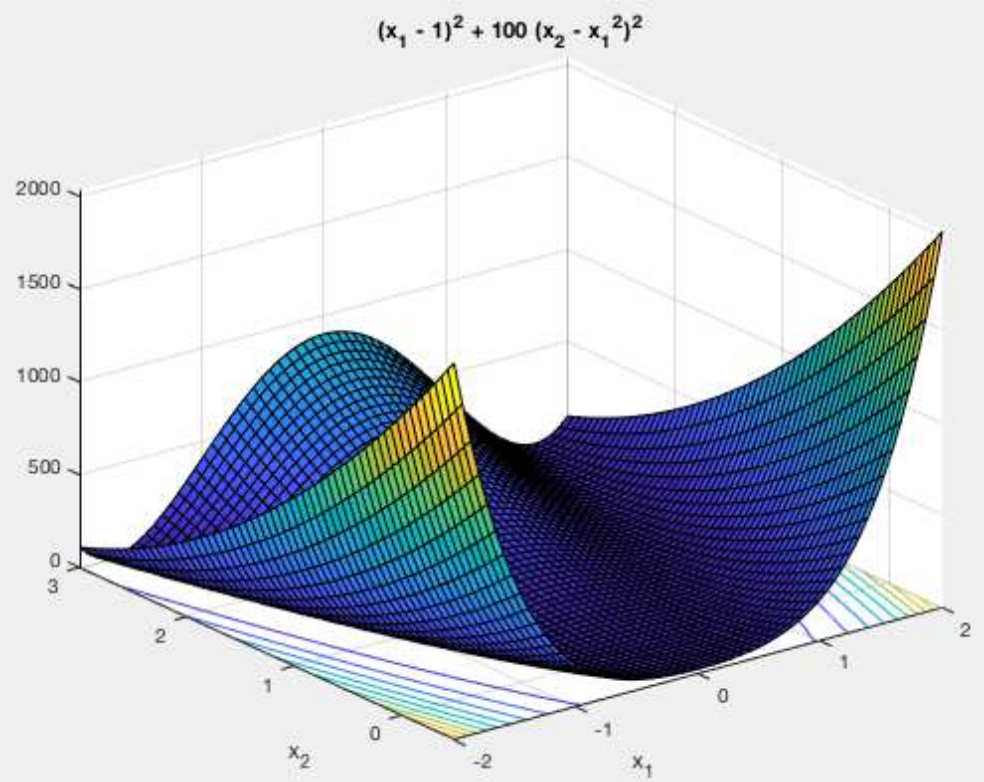


Figure 2.7 Rosenbrock Function (Left) and the Search Paths to Its Minimum (Right) The search paths for Newton's and gradient descent methods are shown in red and blue respectively.

imum from an initial guess at $(-0.5,\ 1.5)$. While it takes only a few iteration steps for the Newton's method to converge to the minimum, following the search path shown in red in the right panel of Fig. 2.7, it takes many more steps for the gradient descent method to converge to the minimum, following the search path shown in blue in the right panel. Also, Fig. 2.8 shows a zoomed-in version of the last stage of the search path toward the final solution. We see that when the step size δ is small, the zigzag search path is composed of a long sequence consecutive segments perpendicular to each other.

2.5 Quasi-Newton Methods

As we have seen above, Newton's methods can be used to solve both nonlinear systems to find roots of a set of simultaneous equations, and optimization problems to minimize a scalar-valued objective function based on the iterations of the same form. Specifically,

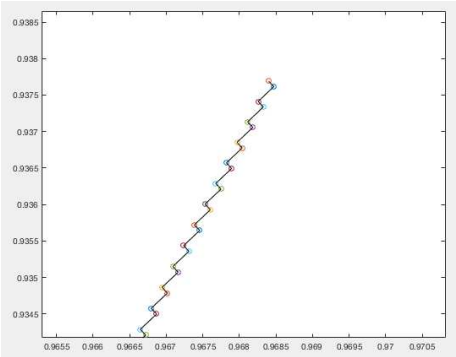


Figure 2.8 Search Path of the Gradient Descent (zoomed-in)

- Solving equations $\mathbf{f}(\mathbf{x}) = \mathbf{0}$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}(\mathbf{x}_n)^{-1}\mathbf{f}(\mathbf{x}_n) = \mathbf{x}_n - \mathbf{J}_n^{-1}\mathbf{f}_n \tag{2.57}$$

  where $\mathbf{J}_n = \mathbf{J}(\mathbf{x}_n)$ is the Jacobian matrix of $\mathbf{f}(\mathbf{x}_n)$,
- Minimizing $f(\mathbf{x})$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}(\mathbf{x}_n)^{-1}\mathbf{g}(\mathbf{x}_n) = \mathbf{x}_n - \mathbf{H}_n^{-1}\mathbf{g}_n \tag{2.58}$$

  where $\mathbf{g}_n = \mathbf{g}(\mathbf{x}_n)$ and $\mathbf{H}_n = \mathbf{H}(\mathbf{x}_n)$ are the gradient vector and Hessian matrix of $f(\mathbf{x})$ at $\mathbf{x}_n$, respectively.

If the Jacobian $\mathbf{J}(\mathbf{x})$ or the Hessian matrix $\mathbf{H}(\mathbf{x})$ is not available, or if it is too computationally costly to calculate its inverse (with complexity $O(N^3)$), the quasi-Newton methods can be used to approximate the Hessian matrix or its inverse based only on the first order derivative, the gradient $\mathbf{g}$ of $f(\mathbf{x})$ (with complexity $O(N^2)$), similar to Broyden's method previously considered (Section 1.5).

In the following, we consider the minimization of a function $f(\mathbf{x})$. Its Taylor expansion around point $\mathbf{x}_{n+1}$ is

$$f(\mathbf{x}) = f(\mathbf{x}_{n+1}) + (\mathbf{x}-\mathbf{x}_{n+1})^T\mathbf{g}_{n+1} + \frac{1}{2}(\mathbf{x}-\mathbf{x}_{n+1})^T\mathbf{H}_{n+1}(\mathbf{x}-\mathbf{x}_{n+1}) + O(||\mathbf{x}-\mathbf{x}_{n+1}||^3) \tag{2.59}$$

Taking derivative with respect to $\mathbf{x}$, we get

$$\frac{d}{d\mathbf{x}}f(\mathbf{x}) = \mathbf{g}(\mathbf{x}) = \mathbf{g}_{n+1} + \mathbf{H}_{n+1}(\mathbf{x} - \mathbf{x}_{n+1}) + O(||\mathbf{x} - \mathbf{x}_{n+1}||^2) \tag{2.60}$$

Evaluating at $\mathbf{x} = \mathbf{x}_n$, we have $\mathbf{g}(\mathbf{x}_n) = \mathbf{g}_n$, and the above can be written as

$$\begin{aligned} \mathbf{g}_{n+1} - \mathbf{g}_n &= \mathbf{H}_{n+1}(\mathbf{x}_{n+1} - \mathbf{x}_n) + O(||\mathbf{x}_{n+1} - \mathbf{x}_n||^2) \\ &= \mathbf{B}_{n+1}(\mathbf{x}_{n+1} - \mathbf{x}_n) \end{aligned} \tag{2.61}$$

where matrix $\mathbf{B}_n$ is the secant approximation of the Hessian matrix $\mathbf{H}_n$, and the

last equality is called the *secant equation*. For convenience, we further define:

$$\mathbf{s}_n = \mathbf{x}_{n+1} - \mathbf{x}_n, \qquad \mathbf{y}_n = \mathbf{g}_{n+1} - \mathbf{g}_n \tag{2.62}$$

so that the equation above can be written as

$$\mathbf{B}_{n+1}\mathbf{s}_n = \mathbf{y}_n, \quad \text{or} \quad \mathbf{B}_{n+1}^{-1}\mathbf{y}_n = \mathbf{s}_n \tag{2.63}$$

This is the *quasi-Newton equation*, which is the *secant condition* that must be satisfied by matrix $\mathbf{B}_n$, or its inverse $\mathbf{B}_n^{-1}$, in any of the quasi-Newton algorithms, all taking the follow general steps:

1. Initialize $\mathbf{x}_0$ and $\mathbf{B}_0$, set $n = 0$;
2. Compute gradient $\mathbf{g}_n$ and the search direction $\mathbf{d}_n = -\mathbf{B}_n^{-1}\mathbf{g}_n$;
3. Get $\mathbf{x}_{n+1} = \mathbf{x}_n + \delta\mathbf{d}_n$ with step size δ satisfying the Wolfe conditions
4. update $\mathbf{B}_{n+1} = \mathbf{B}_n + \Delta\mathbf{B}_n$ or $\mathbf{B}_{n+1}^{-1} = \mathbf{B}_n^{-1} + \Delta\mathbf{B}_n^{-1}$, that satisfies the quasi-Newton equation;
5. If termination condition is not satisfied, $n = n + 1$, go back to second step.

For this iteration to converge to a local minimum of $f(\mathbf{x})$, $\mathbf{B}_n$ must be positive definite matrix, same as the Hessian matrix $\mathbf{H}$ it approximates. Specially, if $\mathbf{B}_n = \mathbf{I}$, then $\mathbf{d}_n = -\mathbf{g}_n$, the algorithm becomes the gradient descent method; also, if $\mathbf{B}_n = \mathbf{H}_n$ is the Hessian matrix, then the algorithm becomes the Newton's method.

In a quasi-Newton method, we can choose to update either $\mathbf{B}_n$ or its inverse $\mathbf{B}_n^{-1}$ based on one of the two forms of the quasi-Newton equation. We note that there is a dual relationship between the two ways to update, i.e., by swapping $\mathbf{y}_n$ and $\mathbf{s}_m$, an update formula for $\mathbf{B}_n$ can be directly applied to its inverse and vice versa.

- The *Symmetric Rank 1 (SR1)* Algorithm

  Here matrix $\mathbf{B}_n$ is updated by an additional term:

  $$\mathbf{B}_{n+1} = \mathbf{B}_n + \mathbf{u}\mathbf{u}^T \tag{2.64}$$

  where $\mathbf{u}$ a vector and $\mathbf{u}\mathbf{u}^T$ is a matrix of rank 1. Imposing the secant condition, we get

  $$\mathbf{B}_{n+1}\mathbf{s}_n = \mathbf{B}_n\mathbf{s}_n + \mathbf{u}\mathbf{u}^T\mathbf{s}_n = \mathbf{y}_n, \quad \text{i.e.,} \quad \mathbf{u}(\mathbf{u}^T\mathbf{s}_n) = \mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n \tag{2.65}$$

  This equation indicates $\mathbf{u}$ is in the same direction as $\mathbf{w} = \mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n$, and can therefore be written as $\mathbf{u} = c\mathbf{w}$. Substituting this back we get

  $$\mathbf{u}\mathbf{u}^T\mathbf{s}_n = c^2\mathbf{w}\mathbf{w}^T\mathbf{s}_n = \mathbf{w} \tag{2.66}$$

  Solving this we get $c = 1/(\mathbf{w}^T\mathbf{s}_n)^{1/2}$,

  $$\mathbf{u} = c\mathbf{w} = \frac{\mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n}{(\mathbf{w}^T\mathbf{s}_n)^{1/2}} = \frac{\mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n}{((\mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n)^T\mathbf{s}_n)^{1/2}} \tag{2.67}$$

and the iteration of $\mathbf{B}_n$ becomes

$$\mathbf{B}_{n+1} = \mathbf{B}_n + \mathbf{u}\mathbf{u}^T = \mathbf{B}_n + \frac{(\mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n)(\mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n)^T}{(\mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n)^T\mathbf{s}_n} \tag{2.68}$$

Applying the Sherman-Morrison formula (Section A.2.4) we further get the inverse $\mathbf{B}_{n+1}$:

$$\begin{aligned}\mathbf{B}_{n+1}^{-1} = \left(\mathbf{B}_n + \mathbf{u}\mathbf{u}^T\right)^{-1} &= \mathbf{B}_n^{-1} - \frac{\mathbf{B}_n^{-1}\mathbf{u}\mathbf{u}^T\mathbf{B}_n^{-1}}{1 + \mathbf{u}^T\mathbf{B}_n^{-1}\mathbf{u}} \\ &= \mathbf{B}_n^{-1} + \frac{(\mathbf{s}_n - \mathbf{B}_n^{-1}\mathbf{y}_n)(\mathbf{s}_n - \mathbf{B}_n^{-1}\mathbf{y}_n)^T}{(\mathbf{s}_n - \mathbf{B}_n^{-1}\mathbf{y}_n)^T\mathbf{y}_n}\end{aligned} \tag{2.69}$$

We note that this equation is dual to the previous one, and it can also be obtained based on the duality between the secant conditions for $\mathbf{B}_n$ and $\mathbf{B}_n^{-1}$. Specifically, same as above, we impose the secant condition for the inverse $\mathbf{B}_{n+1}^{-1}$ of the following form

$$\mathbf{B}_{n+1}^{-1} = \mathbf{B}_n^{-1} + \mathbf{u}\mathbf{u}^T \tag{2.70}$$

and get

$$\mathbf{B}_{n+1}^{-1}\mathbf{y}_n = \mathbf{B}_n^{-1}\mathbf{y}_n + \mathbf{u}\mathbf{u}^T\mathbf{y}_n = \mathbf{s}_n, \tag{2.71}$$

Then by repeating the same process above, we get the same update formula for $\mathbf{B}_n^{-1}$.

This is the formula for directly updating $\mathbf{B}_n^{-1}$. For the iteration to converge to a local minimum, matrix $\mathbf{B}_{n+1}^{-1}$ needs to be positive definite as well as $\mathbf{B}_n^{-1}$, we therefore require

$$\mathbf{w}^T\mathbf{y}_n = (\mathbf{s}_n - \mathbf{B}_n^{-1}\mathbf{y}_n)^T\mathbf{y}_n = \mathbf{s}_n^T\mathbf{y}_n - \mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{y}_n > 0 \tag{2.72}$$

It may be difficult to maintain this requirement through out the iteration. This problem can be avoided in the following DFP and BFGS methods.

- The *BFGS (Broyden-Fletcher-Goldfarb-Shanno)* algorithm

  This is a rank 2 algorithm in which matrix $\mathbf{B}_n$ is updated by two rank-1 terms:

$$\mathbf{B}_{n+1} = \mathbf{B}_n + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T \tag{2.73}$$

  Imposing the secant condition for $\mathbf{B}_n$, we get

$$\mathbf{B}_{n+1}\mathbf{s}_n = (\mathbf{B}_n + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T)\mathbf{s}_n = \mathbf{B}_n\mathbf{s}_n + \mathbf{u}(\alpha\mathbf{u}^T\mathbf{s}_n) + \mathbf{v}(\beta\mathbf{v}^T\mathbf{s}_n) = \mathbf{y}_n \tag{2.74}$$

  or

$$\mathbf{u}(\alpha\mathbf{u}^T\mathbf{s}_n) + \mathbf{v}(\beta\mathbf{v}^T\mathbf{s}_n) = \mathbf{y}_n - \mathbf{B}_n\mathbf{s}_n \tag{2.75}$$

  which can be satisfied if we let

$$\mathbf{u} = \mathbf{y}_n, \qquad \alpha = \frac{1}{\mathbf{s}_n^T\mathbf{y}_n}, \qquad \mathbf{v} = \mathbf{B}_n\mathbf{s}_n, \qquad \beta = -\frac{1}{\mathbf{v}^T\mathbf{s}_n} = -\frac{1}{\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n} \tag{2.76}$$

Substituting these into $\mathbf{B}_{n+1} = \mathbf{B}_n + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T$ we get

$$\mathbf{B}_{n+1} = \mathbf{B}_n + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T = \mathbf{B}_n + \frac{\mathbf{y}_n\mathbf{y}_n^T}{\mathbf{y}_n^T\mathbf{s}_n} - \frac{\mathbf{B}_n\mathbf{s}_n\mathbf{s}_n^T\mathbf{B}_n}{\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n} \tag{2.77}$$

Given this update formula for $\mathbf{B}_n$, we can further find its inverse $\mathbf{B}_n^{-1}$ and then the search direction $\mathbf{d}_n = -\mathbf{B}_n^{-1}\mathbf{g}_n$.

Alternatively, given $\mathbf{B}_n$, we can further derive an update formula directly for the inverse matrix $\mathbf{B}_n^{-1}$, so that the search direction $\mathbf{d}_n$ can be obtained without carrying out the inversion computation. Specifically, we first define $\mathbf{U} = [\mathbf{u}_1\ \mathbf{u}_2]$ and $\mathbf{V} = [\mathbf{v}_1\ \mathbf{v}_2]$, where

$$\mathbf{u}_1 = \mathbf{v}_1 = \frac{\mathbf{y}_n}{(\mathbf{s}_n^T\mathbf{y}_n)^{1/2}}, \qquad \mathbf{u}_2 = -\mathbf{v}_2 = \frac{\mathbf{B}_n\mathbf{s}_n}{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}} \tag{2.78}$$

so that the expression above can be written as:

$$\mathbf{B}_{n+1} = \mathbf{B}_n + \mathbf{u}_1\mathbf{v}_1^T + \mathbf{u}_2\mathbf{v}_2^T = (\mathbf{B}_n + \mathbf{U}\mathbf{V}^T)^{-1} \tag{2.79}$$

and then apply the Sherman-Morrison formula to get:

$$\begin{aligned}\mathbf{B}_{n+1}^{-1} &= (\mathbf{B}_n + \mathbf{U}\mathbf{V}^T)^{-1} = \mathbf{B}_n^{-1} - \mathbf{B}_n^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^T\mathbf{B}_n^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{B}_n^{-1} \\ &= \mathbf{B}_n^{-1} - \mathbf{B}_n^{-1}\mathbf{U}\mathbf{C}^{-1}\mathbf{V}^T\mathbf{B}_n^{-1} \\ &= \mathbf{B}_n^{-1} - \mathbf{B}_n^{-1}[\mathbf{u}_1\ \mathbf{u}_2]\mathbf{C}^{-1}(\mathbf{B}_n^{-1}[\mathbf{v}_1\ \mathbf{v}_2])^T \end{aligned} \tag{2.80}$$

where we have defined

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \mathbf{I} + \mathbf{V}^T\mathbf{B}_n^{-1}\mathbf{U} = \mathbf{I} + [\mathbf{v}_1\ \mathbf{v}_2]^T\mathbf{B}_n^{-1}[\mathbf{u}_1\ \mathbf{u}_2] \tag{2.81}$$

with

$$c_{11} = 1 + \mathbf{v}_1^T\mathbf{B}_n^{-1}\mathbf{u}_1 = 1 + \frac{\mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{y}_n}{\mathbf{s}_n^T\mathbf{y}_n} \tag{2.82}$$

$$c_{22} = 1 + \mathbf{v}_2\mathbf{B}_n^{-1}\mathbf{u}_2 = 1 - \frac{\mathbf{s}_n^T\mathbf{B}_n\mathbf{B}_n^{-1}\mathbf{B}_n\mathbf{s}_n}{\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n} = 0 \tag{2.83}$$

$$c_{12} = \mathbf{v}_1^T\mathbf{B}_n^{-1}\mathbf{u}_2 = \frac{\mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{B}_n\mathbf{s}_n}{(\mathbf{s}_n^T\mathbf{y}_n)^{1/2}(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}} = \frac{(\mathbf{s}_n^T\mathbf{y}_n)^{1/2}}{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}} \tag{2.84}$$

$$c_{21} = \mathbf{v}_2^T\mathbf{B}_n^{-1}\mathbf{u}_1 = -c_{12} \tag{2.85}$$

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} \\ -c_{12} & 0 \end{bmatrix}, \qquad \mathbf{C}^{-1} = \begin{bmatrix} 0 & -1/c_{12} \\ 1/c_{12} & c_{11}/c_{12}^2 \end{bmatrix} \tag{2.86}$$

Substituting this $\mathbf{C}^{-1}$ into the expression for $\mathbf{B}_{n+1}^{-1}$, we get:

$$\begin{aligned}\mathbf{B}_{n+1}^{-1} &= \mathbf{B}_n^{-1} - [\mathbf{B}_n^{-1}\mathbf{u}_1\ \mathbf{B}_n^{-1}\mathbf{u}_2]\begin{bmatrix} 0 & -1/c_{12} \\ 1/c_{12} & c_{11}/c_{12}^2 \end{bmatrix}[\mathbf{B}_n^{-1}\mathbf{v}_1\ \mathbf{B}_n^{-1}\mathbf{v}_2]^T \\ &= \mathbf{B}_n^{-1} - \frac{1}{c_{12}}[\mathbf{B}_n^{-1}\mathbf{u}_2\mathbf{v}_1^T\mathbf{B}_n^{-1} - \mathbf{B}_n^{-1}\mathbf{u}_1\mathbf{v}_2^T\mathbf{B}_n^{-1}] - \frac{c_{11}}{c_{12}^2}\mathbf{B}_n^{-1}\mathbf{u}_2\mathbf{v}_2^T\mathbf{B}_n^{-1} \end{aligned} \tag{2.87}$$

where

$$\frac{1}{c_{12}}\mathbf{B}_n^{-1}\mathbf{u}_2\mathbf{v}_1^T\mathbf{B}_n^{-1} = \frac{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}}{(\mathbf{s}_n^T\mathbf{y}_n)^{1/2}}\mathbf{B}_n^{-1}\frac{\mathbf{B}_n\mathbf{s}_n}{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}}\frac{\mathbf{y}_n^T}{(\mathbf{s}_n^T\mathbf{y}_n)^{1/2}}\mathbf{B}_n^{-1} = \frac{\mathbf{s}_n\mathbf{y}_n^T\mathbf{B}_n^{-1}}{\mathbf{s}_n^T\mathbf{y}_n} \tag{2.88}$$

$$-\frac{1}{c_{12}}\mathbf{B}_n^{-1}\mathbf{u}_1\mathbf{v}_2^T\mathbf{B}_n^{-1} = \frac{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}}{(\mathbf{s}_n^T\mathbf{y}_n)^{1/2}}\mathbf{B}_n^{-1}\frac{\mathbf{y}_n}{(\mathbf{s}_n^T\mathbf{y}_n)^{1/2}}\frac{\mathbf{s}_n^T\mathbf{B}_n}{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}}\mathbf{B}_n^{-1} = \frac{\mathbf{B}_n^{-1}\mathbf{y}_n\mathbf{s}_n^T}{\mathbf{s}_n^T\mathbf{y}_n} \tag{2.89}$$

$$\begin{aligned}\frac{c_{11}}{c_{12}^2}\mathbf{B}_n^{-1}\mathbf{u}_2\mathbf{v}_2^T\mathbf{B}_n^{-1} &= -\left(1+\frac{\mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{y}_n}{\mathbf{s}_n^T\mathbf{y}_n}\right)\frac{\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n}{\mathbf{s}_n^T\mathbf{y}_n}\mathbf{B}_n^{-1}\frac{\mathbf{B}_n\mathbf{s}_n}{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}}\frac{\mathbf{s}_n^T\mathbf{B}_n}{(\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n)^{1/2}}\mathbf{B}_n^{-1} \\ &= -\left(1+\frac{\mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{y}_n}{\mathbf{s}_n^T\mathbf{y}_n}\right)\frac{\mathbf{s}_n\mathbf{s}_n^T}{\mathbf{s}_n^T\mathbf{y}_n}\end{aligned} \tag{2.90}$$

Substituting these three terms back into Eq. (2.87), we get the update formula for $\mathbf{B}_n^{-1}$:

$$\mathbf{B}_{n+1}^{-1} = \mathbf{B}_n^{-1} - \frac{\mathbf{B}_n^{-1}\mathbf{y}_n\mathbf{s}_n^T + \mathbf{s}_n\mathbf{y}_n^T\mathbf{B}_n^{-1}}{\mathbf{s}_n^T\mathbf{y}_n} + \left(1+\frac{\mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{y}_n}{\mathbf{s}_n^T\mathbf{y}_n}\right)\frac{\mathbf{s}_n\mathbf{s}_n^T}{\mathbf{s}_n^T\mathbf{y}_n} \tag{2.91}$$

In summary, $\mathbf{B}_{n+1}^{-1}$ can be found by either Eq. (2.77) for $\mathbf{B}_{n+1}$ followed by an additional inversion operation, or Eq. (2.91) directly for $\mathbf{B}_{n+1}^{-1}$. The results obtained by these methods are equivalent.

- The *DFP (Davidon-Fletcher-Powell)* Algorithm

  Same as the BFGS algorithm, the DFP algorithm is also a rank 2 algorithm, where instead of $\mathbf{B}_n$, the inverse matrix $\mathbf{B}_n^{-1}$ is updated by two rank-1 terms:

$$\mathbf{B}_{n+1}^{-1} = \mathbf{B}_n^{-1} + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T \tag{2.92}$$

  where α and β are real scalars, $\mathbf{u}$ and $\mathbf{v}$ are vectors. Imposing the secant condition for the inverse of $\mathbf{B}_n$, we get

$$\mathbf{B}_{n+1}^{-1}\mathbf{y}_n = (\mathbf{B}_n^{-1}+\alpha\mathbf{u}\mathbf{u}^T+\beta\mathbf{v}\mathbf{v}^T)\mathbf{y}_n = \mathbf{B}_n^{-1}\mathbf{y}_n+\mathbf{u}(\alpha\mathbf{u}^T\mathbf{y}_n)+\mathbf{v}(\beta\mathbf{v}^T\mathbf{y}_n) = \mathbf{s}_n \tag{2.93}$$

  or

$$\mathbf{u}(\alpha\mathbf{u}^T\mathbf{y}_n) + \mathbf{v}(\beta\mathbf{v}^T\mathbf{y}_n) = \mathbf{s}_n - \mathbf{B}_n^{-1}\mathbf{y}_n \tag{2.94}$$

  which can be satisfied if we let

$$\mathbf{u} = \mathbf{s}_n, \qquad \alpha = \frac{1}{\mathbf{s}^T\mathbf{y}_n}, \qquad \mathbf{v} = \mathbf{B}_n^{-1}\mathbf{y}_n, \qquad \beta = -\frac{1}{\mathbf{v}^T\mathbf{y}_n} = -\frac{1}{\mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{y}_n} \tag{2.95}$$

  Substituting these into $\mathbf{B}_{n+1}^{-1} = \mathbf{B}_n^{-1} + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T$ we get

$$\mathbf{B}_{n+1}^{-1} = \mathbf{B}_n^{-1} + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T = \mathbf{B}_n^{-1} + \frac{\mathbf{s}_n\mathbf{s}_n^T}{\mathbf{s}_n^T\mathbf{y}_n} - \frac{\mathbf{B}_n^{-1}\mathbf{y}_n\mathbf{y}_n^T\mathbf{B}_n^{-1}}{\mathbf{y}_n^T\mathbf{B}_n^{-1}\mathbf{y}_n} \tag{2.96}$$

  Following the same procedure used for the BFGS method, we can also

obtain the update formula of matrix $\mathbf{B}_n$ for the DFP method:

$$\mathbf{B}_{n+1} = \mathbf{B}_n - \frac{\mathbf{B}_n \mathbf{s}_n \mathbf{y}_n^T + \mathbf{y}_n \mathbf{s}_n^T \mathbf{B}_n}{\mathbf{y}_n^T \mathbf{s}_n} + \left(1 + \frac{\mathbf{s}_n^T \mathbf{B}_n \mathbf{s}_n}{\mathbf{y}_n^T \mathbf{s}_n}\right) \frac{\mathbf{y}_n \mathbf{y}_n^T}{\mathbf{y}_n^T \mathbf{s}_n} \tag{2.97}$$

We note that Eqs. (2.96) and (2.77) form a duality pair, and Eqs. (2.97) and (2.91) form another duality pair. In other words, the BFGS and FDP methods are dual of each other.

In both the DFP and BFGS methods, matrix $\mathbf{B}_n^{-1}$, as well as $\mathbf{B}_n$, must be positive definite, i.e., $\mathbf{z}^T \mathbf{B}_n \mathbf{z} > 0$ must hold for any $\mathbf{z} \neq \mathbf{0}$. We now prove that this requirement is satisfied if the curvature condition of the Wolfe conditions is satisfied:

$$\mathbf{g}_{n+1}^T \mathbf{d}_n \geq c_2 \, \mathbf{g}_n^T \mathbf{d}_n \qquad \text{i.e.} \qquad (\mathbf{g}_{n+1} - c_2 \, \mathbf{g}_n)^T \, \mathbf{d}_n \geq 0 \tag{2.98}$$

Replacing the search direction $\mathbf{d}_n$ by $\delta \mathbf{d}_n = \mathbf{x}_{n+1} - \mathbf{x}_n = \mathbf{s}_n$, we get:

$$(\mathbf{g}_{n+1} - c_2 \mathbf{g}_n)^T \mathbf{s}_n = \mathbf{g}_{n+1}^T \mathbf{s}_n - c_2 \mathbf{g}_n^T \mathbf{s}_n \geq 0 \tag{2.99}$$

We also have

$$\mathbf{y}_n^T \mathbf{s}_n = (\mathbf{g}_{n+1} - \mathbf{g}_n)^T \mathbf{s}_n = \mathbf{g}_{n+1}^T \mathbf{s}_n - \mathbf{g}_n^T \mathbf{s}_n \tag{2.100}$$

Subtracting the first equation from the second, we get

$$\mathbf{y}_n^T \mathbf{s}_n - (\mathbf{g}_{n+1}^T \mathbf{s}_n - c_2 \mathbf{g}_n^T \mathbf{s}_n) = (c_2 - 1) \mathbf{g}_n^T \mathbf{s}_n > 0 \tag{2.101}$$

The last inequality is due to the fact that$\mathbf{g}_n^T \mathbf{s}_n < 0$ and $c_2 < 1$. We therefore have

$$\mathbf{y}_n^T \mathbf{s}_n \geq \mathbf{g}_{n+1}^T \mathbf{s}_n - c_2 \mathbf{g}_n^T \mathbf{s}_n \geq 0 \tag{2.102}$$

Given $\mathbf{y}_n^T \mathbf{s}_n > 0$, we can further prove by induction that both $\mathbf{B}_{n+1}$ and $\mathbf{B}_{n+1}^{-1}$ based respectively on the update formulae in Eqs. (2.77) and (2.96) are positive definite. We first assume $\mathbf{B}_0$ and $\mathbf{B}_n$ are both positive definite, and express $\mathbf{B}_n$ in terms of its Cholesky decomposition, $\mathbf{B}_n = \mathbf{L}\mathbf{L}^T$. We further define $\mathbf{a} = \mathbf{L}^T \mathbf{z}$, $\mathbf{b} = \mathbf{L}^T \mathbf{s}$, and get

$$\mathbf{a}^T \mathbf{a} = \mathbf{z}^T \mathbf{B}_n \mathbf{z}, \quad \mathbf{b}^T \mathbf{b} = \mathbf{s}^T \mathbf{B}_n \mathbf{s}, \quad \mathbf{a}^T \mathbf{b} = \mathbf{z}^T \mathbf{B}_n \mathbf{s} \tag{2.103}$$

Now we can show that the sum of the first and third terms of Eq. (2.91) is positive definite:

$$\mathbf{z}^T \left[\mathbf{B}_n - \frac{\mathbf{B}_n \mathbf{s}_n \mathbf{s}_n^T \mathbf{B}_n}{\mathbf{s}_n^T \mathbf{B}_n \mathbf{s}_n}\right] \mathbf{z} = \mathbf{z}^T \mathbf{B}_n \mathbf{z} - \frac{(\mathbf{z}^T \mathbf{B}_n \mathbf{s}_n)^2}{\mathbf{s}_n^T \mathbf{B}_n \mathbf{s}_n} = \mathbf{a}^T \mathbf{a} - \frac{(\mathbf{a}^T \mathbf{b})^2}{\mathbf{b}^T \mathbf{b}} \geq 0 \tag{2.104}$$

The last step is due to the Cauchy-Schwarz inequality. Also, as $\mathbf{s}_n^T \mathbf{y}_n \geq 0$, the second term of Eq. (2.77) is positive definite

$$\mathbf{z}^T \left(\frac{\mathbf{s}_n \mathbf{s}_n^T}{\mathbf{s}_n^T \mathbf{y}_n}\right) \mathbf{z} \geq 0 \tag{2.105}$$

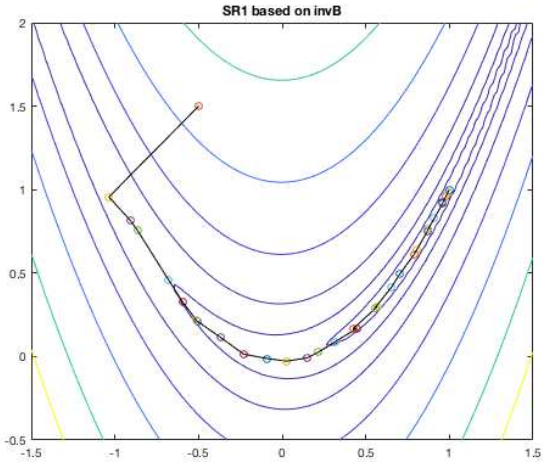


Figure 2.9 Search Path of SR1 based on $\mathbf{B}^{-1}$

Combining these two results, we get

$$\mathbf{z}^T\mathbf{B}_{n+1}\mathbf{z} = \mathbf{z}^T\left(\mathbf{B}_n - \frac{\mathbf{B}_n\mathbf{s}_n\mathbf{s}_n^T\mathbf{B}_n}{\mathbf{s}_n^T\mathbf{B}_n\mathbf{s}_n}\right)\mathbf{z} + \mathbf{z}^T\left(\frac{\mathbf{y}_n\mathbf{y}_n^T}{\mathbf{s}_n^T\mathbf{y}_n}\right)\mathbf{z} \geq 0 \tag{2.106}$$

i.e., $\mathbf{B}_{n+1}$ based on the update formula Eq. (2.77) is positive definite. Following the same steps, we can also show that $\mathbf{B}_{n+1}^{-1}$ based on Eq. (2.96) is positive definite.

As examples, Figs. 2.9, 2.10 and 2.11 show respectively the search path of the SR1, DFP, and BFGS methods when applied to find the minimum of the Rosenbrock function.

2.6 Conjugate gradient method

The gradient descent method can be used to solve the minimization problem when the Hessian matrix of the objective function is not available. However, this method may be inefficient if it gets into a zigzag search pattern and repeat the same search directions many times. This problem can be avoided by the *conjugate gradient (CG)* method. If the objective function is quadratic, the CG method converges to the solution in N iterations without repeating any of the directions previously traversed. If the objective function is not quadratic, the CG method can still significantly improve the performance in comparison to the gradient descent method.

Again consider the approximation of the function $f(\mathbf{x})$ to be minimized by the

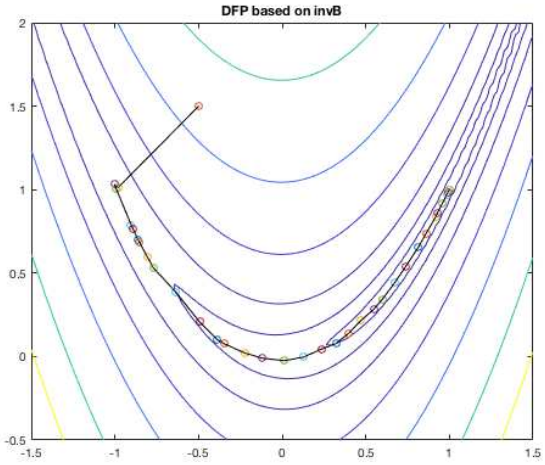


Figure 2.10 Search Path of DFP based on $\mathbf{B}^{-1}$

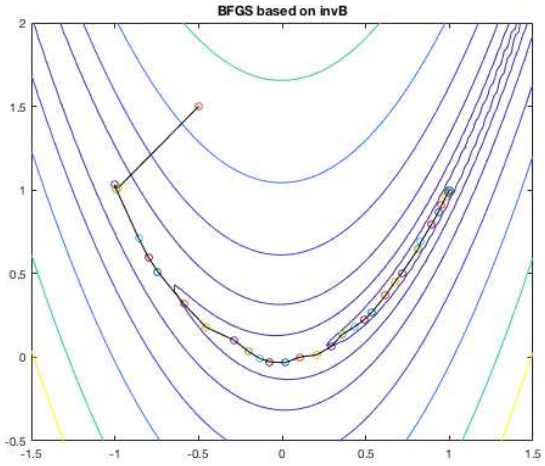


Figure 2.11 Search Path of BFGS based on $\mathbf{B}^{-1}$

first three terms of its Taylor series:

$$f(\mathbf{x}) = f(\mathbf{x}_0 + \delta\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{g}_0^T \delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^T \mathbf{H}_0 \delta\mathbf{x} \tag{2.107}$$

If the Hessian matrix is positive definite, then $f(\mathbf{x})$ has a minimum. If function $f(\mathbf{x})$ is quadratic, the approximation above becomes exact and the function can be written as

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A}\mathbf{x} - \mathbf{b}^T \mathbf{x} + c \tag{2.108}$$

where $\mathbf{A} = \mathbf{H}$ is the symmetric Hessian matrix, and its gradient and Hessian can be written as the following respectively:

$$\mathbf{g}(\mathbf{x}) = \frac{d}{d\mathbf{x}} f(\mathbf{x}) = \frac{d}{d\mathbf{x}} \left(\frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + c \right) = \mathbf{A}\mathbf{x} - \mathbf{b} \tag{2.109}$$

and

$$\mathbf{H}(\mathbf{x}) = \frac{d^2}{d\mathbf{x}^2} f(\mathbf{x}) = \frac{d}{d\mathbf{x}} \mathbf{g} = \frac{d}{d\mathbf{x}} (\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{A} \tag{2.110}$$

Solving $\mathbf{g}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$, we get the solution $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$, at which the function is minimized to

$$f(\mathbf{x}^*) = \frac{1}{2} (\mathbf{A}^{-1}\mathbf{b})^T \mathbf{A} (\mathbf{A}^{-1}\mathbf{b}) - \mathbf{b}^T (\mathbf{A}^{-1}\mathbf{b}) + c = -\frac{1}{2} \mathbf{b}^T \mathbf{A}^{-1} \mathbf{b} + c \tag{2.111}$$

At the solution $\mathbf{x}^*$ that minimizes $f(\mathbf{x})$, its gradient is

$$\mathbf{g}(\mathbf{x}^*) = \mathbf{A}\mathbf{x}^* - \mathbf{b} = \mathbf{0} \tag{2.112}$$

We also see that the minimization of the quadratic function $f(\mathbf{x})$ is equivalent to solving a linear equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ with a symmetric positive definite coefficient matrix $\mathbf{A}$. The CG method considered here can therefore be used for solving both problems.

Conjugate basis vectors

We first review the concept of *conjugate vectors* (Section A.1.2) which is of essential importance in the CG method. Two vectors $\mathbf{u}$ and $\mathbf{v}$ are *mutually conjugate* or *A-conjugate* to each other with respect to a symmetric matrix $\mathbf{A} = \mathbf{A}^T$, if they satisfy:

$$\mathbf{u}^T \mathbf{A} \mathbf{v} = (\mathbf{A}\mathbf{v})^T \mathbf{u} = \mathbf{v}^T \mathbf{A} \mathbf{u} = 0 \tag{2.113}$$

Specially, if $\mathbf{A} = \mathbf{I}$, the two conjugate vectors become orthogonal to each other, i.e., $\mathbf{v}^T \mathbf{u} = 0$.

Similar to a set of N orthogonal vectors that can be used as the basis spanning an N-D space, a set of N mutually conjugate vectors $\{\mathbf{d}_0, \cdots, \mathbf{d}_{N-1}\}$ satisfying $\mathbf{d}_i^T \mathbf{A} \mathbf{d}_j = 0$ $(i \neq j)$ can also be used as a basis to span the N-D space. Any vector in the space can be expressed as a linear combination of these basis vectors.

Also we note that any set of N independent vectors can be converted by the Gram-Schmidt process (Section A.1.2) to a set of N basis vectors that are either orthogonal or A-conjugate to each other.

Example 2.6 Given two independent basis vectors of the 2-D space, and a positive-definite matrix:

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$$

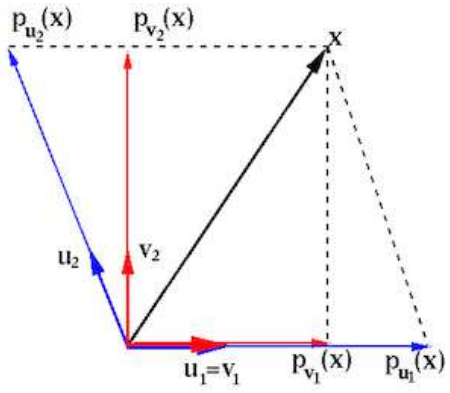


Figure 2.12 A-Orthogonality based on Matrix **A**

we can construct two A-orthogonal basis vectors by the Gram-Schmidt method, as shown in Fig. 2.12.

$$\mathbf{u}_1 = \mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad \mathbf{u}_2 = \mathbf{v}_2 - \frac{\mathbf{u}_1^T \mathbf{A}\mathbf{v}_2}{\mathbf{u}_1^T \mathbf{A}\mathbf{u}_1}\mathbf{u}_1 = \begin{bmatrix} -1/3 \\ 1 \end{bmatrix}$$

The projections of a vector $\mathbf{x} = [2,\ 3]^T$ onto $\mathbf{v}_1$ and $\mathbf{v}_2$ are:

$$\mathbf{p}_{\mathbf{v}_1}(\mathbf{x}) = \frac{\mathbf{v}_1^T \mathbf{x}}{\mathbf{v}_1^T \mathbf{v}_1}\mathbf{v}_1 = 2\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, \quad \mathbf{p}_{\mathbf{v}_2}(\mathbf{x}) = \frac{\mathbf{v}_2^T \mathbf{x}}{\mathbf{v}_2^T \mathbf{v}_2}\mathbf{v}_2 = 3\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

The A-projections of the same vector $\mathbf{x}$ onto $\mathbf{u}_1$ and $\mathbf{u}_2$ are:

$$\mathbf{p}_{\mathbf{u}_1}(\mathbf{x}) = \frac{\mathbf{u}_1^T \mathbf{A}\mathbf{x}}{\mathbf{u}_1^T \mathbf{A}\mathbf{u}_1}\mathbf{u}_1 = 3\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}, \qquad \mathbf{p}_{\mathbf{u}_2}(\mathbf{x}) = \frac{\mathbf{u}_2^T \mathbf{A}\mathbf{x}}{\mathbf{u}_2^T \mathbf{A}\mathbf{u}_2}\mathbf{u}_2 = 3\begin{bmatrix} -1/3 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix}$$

The original vector $\mathbf{x}$ can be represented in either of the two bases:

$$\begin{aligned} \mathbf{x} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} &= 2\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 3\begin{bmatrix} 0 \\ 1 \end{bmatrix} = 2\mathbf{v}_1 + 3\mathbf{v}_2 = \mathbf{p}_{\mathbf{v}_1}(\mathbf{x}) + \mathbf{p}_{\mathbf{v}_2}(\mathbf{x}) \\ &= 3\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 3\begin{bmatrix} -1/3 \\ 1 \end{bmatrix} = 3\mathbf{u}_1 + 3\mathbf{u}_2 = \mathbf{p}_{\mathbf{u}_1}(\mathbf{x}) + \mathbf{p}_{\mathbf{u}_2}(\mathbf{x}) \end{aligned}$$

Search along a conjugate basis

Similar to the gradient descent method, which iteratively improves the estimated solution by following a sequence of orthogonal search directions $\{\mathbf{d}_0, \cdots, \mathbf{d}_n, \cdots\}$ with $\mathbf{d}_n = -\mathbf{g}_n$ satisfying $\mathbf{g}_n^T \mathbf{g}_{n+1} = 0$, the CG method also follows a sequence of search directions $\{\mathbf{d}_0, \cdots, \mathbf{d}_{N-1}\}$ A-orthogonal to each other, i.e., $\mathbf{d}_i^T \mathbf{A}\mathbf{d}_j = 0\ (i \neq j)$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \delta_n \mathbf{d}_n = \cdots = \mathbf{x}_0 + \sum_{i=0}^{n} \delta_i \mathbf{d}_i \tag{2.114}$$

We define the error at the nth step as $\mathbf{e}_n = \mathbf{x}_n - \mathbf{x}^*$. Then subtracting $\mathbf{x}^*$ from both sides, we get the iteration in terms of the errors:

$$\mathbf{e}_{n+1} = \mathbf{e}_n + \delta_n \mathbf{d}_n = \cdots = \mathbf{e}_0 + \sum_{i=0}^{n} \delta_i \mathbf{d}_i \tag{2.115}$$

Due to $\mathbf{g} = \mathbf{A}\mathbf{x} - \mathbf{b}$ in Eq. (2.112), we can find the gradient at the nth step $\mathbf{x}_n$ as

$$\mathbf{g}_n = \mathbf{A}\mathbf{x}_n - \mathbf{b} = \mathbf{A}(\mathbf{x}^* + \mathbf{e}_n) - \mathbf{b} = \mathbf{A}\mathbf{x}^* - \mathbf{b} + \mathbf{A}\mathbf{e}_n = \mathbf{A}\mathbf{e}_n \tag{2.116}$$

As the gradient at the solution is zero $\mathbf{g}(\mathbf{x}^*) = \mathbf{0}$, we can consider the gradient $\mathbf{g}_n$ at $\mathbf{x}_n$ as the residual of the nth iteration, and $\varepsilon = ||\mathbf{g}_n||^2$ an error measurement representing how close $\mathbf{x}_n$ is to the true solution $\mathbf{x}^*$.

The optimal step size given in Eq. (2.39) can now be written as

$$\begin{aligned}\delta_i &= -\frac{\mathbf{d}_i^T \mathbf{g}_i}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i} = -\frac{\mathbf{d}_i^T \mathbf{A}\mathbf{e}_i}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i} = -\frac{\mathbf{d}_i^T \mathbf{A}\left(\mathbf{e}_0 + \sum_{j=0}^{i-1} \delta_j \mathbf{d}_n\right)}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i} \\ &= -\frac{\mathbf{d}_i^T \mathbf{A}\mathbf{e}_0 + \sum_{j=0}^{n-1} \delta_j \mathbf{d}_i^T \mathbf{A}\mathbf{d}_j}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i} = -\frac{\mathbf{d}_i^T \mathbf{A}\mathbf{e}_0}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i}\end{aligned} \tag{2.117}$$

The last equality is due to the fact that $\mathbf{d}_i$ and $\mathbf{d}_j$ are A-orthogonal, $\mathbf{d}_i^T \mathbf{A}\mathbf{d}_j = 0$. Substituting this into Eq. (2.115) we get

$$\mathbf{e}_{n+1} = \mathbf{e}_n + \delta_n \mathbf{d}_n = \mathbf{e}_n - \left(\frac{\mathbf{d}_n^T \mathbf{A}\mathbf{e}_0}{\mathbf{d}_n^T \mathbf{A}\mathbf{d}_n}\right)\mathbf{d}_n \tag{2.118}$$

On the other hand, we can represent the error $\mathbf{e}_0 = \mathbf{x}_0 - \mathbf{x}^*$ associated with the initial guess $\mathbf{x}_0$ as a linear combination of the A-orthogonal search vectors $\{\mathbf{d}_0, \cdots, \mathbf{d}_{N-1}\}$ as N basis vectors that span the N-D vector space:

$$\mathbf{e}_0 = \sum_{i=0}^{N-1} c_i \mathbf{d}_i = \sum_{i=0}^{N-1} \mathbf{p}_{\mathbf{d}_i}(\mathbf{e}_0) = \sum_{i=0}^{N-1} \left(\frac{\mathbf{d}_i^T \mathbf{A}\mathbf{e}_0}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i}\right)\mathbf{d}_i \tag{2.119}$$

where $\mathbf{p}_{\mathbf{d}_i}(\mathbf{e}_0)$ is the A-projection of $\mathbf{e}_0$ onto the ith basis vector $\mathbf{d}_i$:

$$\mathbf{p}_{\mathbf{d}_i}(\mathbf{e}_0) = c_i \mathbf{d}_i = \left(\frac{\mathbf{d}_i^T \mathbf{A}\mathbf{e}_0}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i}\right)\mathbf{d}_i \tag{2.120}$$

Note that the coefficient c_i happens to be the negative optimal step size in Eq. (2.117):

$$c_i = \frac{\mathbf{d}_i^T \mathbf{A}\mathbf{e}_0}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i} = -\delta_i \tag{2.121}$$

Now the expression of $\mathbf{e}_{n+1}$ in Eq. (2.115) can be written as

$$\mathbf{e}_{n+1} = \mathbf{e}_0 + \sum_{i=0}^{n} \delta_i \mathbf{d}_i = \sum_{i=0}^{N-1} c_i \mathbf{d}_i - \sum_{i=0}^{n} c_i \mathbf{d}_i = \sum_{i=n+1}^{N-1} c_i \mathbf{d}_i = \sum_{i=n+1}^{N-1} \mathbf{p}_{\mathbf{d}_i}(\mathbf{e}_0) \tag{2.122}$$

We see that in each iteration, the number of terms in the summation for $\mathbf{e}_0$ is

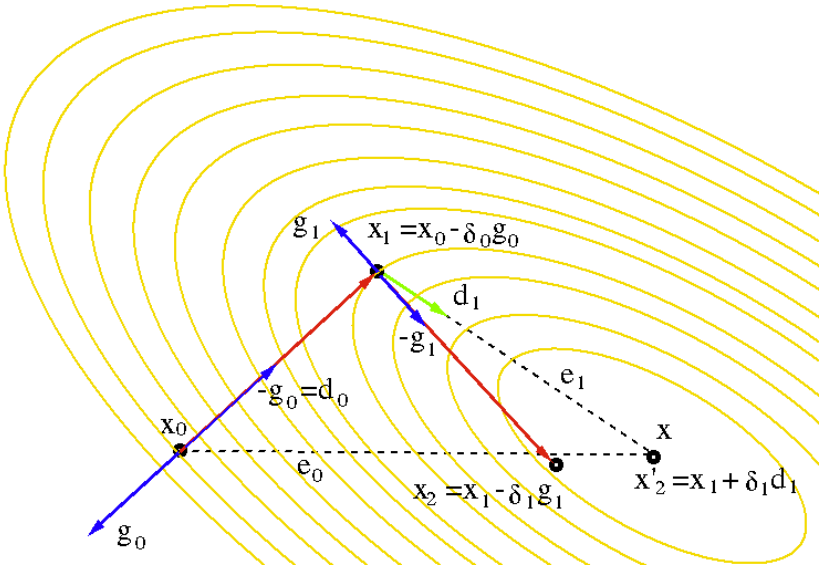


Figure 2.13 Conjugate Gradient Descent

reduced by one, i.e., the nth component $\mathbf{p}_{\mathbf{d}_n}(\mathbf{e}_0)$ of $\mathbf{e}_0$ along the direction of $\mathbf{d}_n$ is completely eliminated. After N such iterations, the error is reduced from $\mathbf{e}_0$ to $\mathbf{e}_N = \mathbf{0}$, and the true solution is obtained $\mathbf{x}_N = \mathbf{x}^* + \mathbf{e}_N = \mathbf{x}^*$.

Pre-multiplying $\mathbf{d}_k^T\mathbf{A}$ $(k \leq n)$ on both sides of the equation above, we get

$$\mathbf{d}_k^T\mathbf{A}\mathbf{e}_{n+1} = \sum_{i=n+1}^{N-1} c_i\mathbf{d}_k^T\mathbf{A}\mathbf{d}_j = 0 \tag{2.123}$$

We see that after $n+1$ iterations the remaining error $\mathbf{e}_{n+1}$ is A-orthogonal to all previous directions $\mathbf{d}_0, \cdots, \mathbf{d}_n$.

Due to Eq. (2.116), the equation above can also be written as

$$\mathbf{d}_k^T\mathbf{A}\mathbf{e}_{n+1} = \mathbf{d}_k^T\mathbf{g}_{n+1} = 0 \tag{2.124}$$

i.e., the gradient $\mathbf{g}_{n+1}$ is orthogonal to all previous search directions.

The conjugate gradient method is compared with the gradient descent method for the case of $N = 2$ in Fig. 2.13. We see that the first search direction is the same $-\mathbf{g}_0$ for both methods, but the next search direction $\mathbf{d}_1$ is A-orthogonal to $\mathbf{d}_0$, same as the next error $\mathbf{e}_1$, different from the search direction $-\mathbf{g}_1$ in gradient descent method. The conjugate gradient method finds the solution $\mathbf{x}$ in $N = 2$ steps, while the gradient descent method has to go through many more steps all orthogonal to each other before it finds the solution.

Find the A-orthogonal basis

The N A-orthogonal search directions $\{\mathbf{d}_0, \cdots, \mathbf{d}_{N-1}\}$ can be constructed based on any set of N independent vectors $\{\mathbf{v}_0, \cdots, \mathbf{v}_{N-1}\}$ by the Gram-Schmidt pro-

cess:

$$\mathbf{d}_n = \mathbf{v}_n - \sum_{j=0}^{n-1} \mathbf{p}_{\mathbf{d}_j}(\mathbf{v}_n) = \mathbf{v}_n - \sum_{m=0}^{n-1} \left(\frac{\mathbf{d}_m^T \mathbf{A} \mathbf{v}_n}{\mathbf{d}_m^T \mathbf{A} \mathbf{d}_m} \right) \mathbf{d}_m = \mathbf{v}_n - \sum_{m=0}^{n-1} \beta_{nm} \mathbf{d}_m \quad (2.125)$$

where $\beta_{nm} = \mathbf{d}_m^T \mathbf{A} \mathbf{v}_n / (\mathbf{d}_m^T \mathbf{A} \mathbf{d}_m)$ and $\beta_{nm} \mathbf{d}_m$ is the A-projection of $\mathbf{v}_n$ onto each of the previous direction $\mathbf{d}_m$.

We will gain some significant computational advantage if we choose to use $\mathbf{v}_n = -\mathbf{g}_n$. Now the Gram-Schmidt process above becomes:

$$\mathbf{d}_n = -\mathbf{g}_n - \sum_{m=0}^{n-1} \beta_{nm} \mathbf{d}_m, \quad (2.126)$$

where

$$\beta_{nm} = \frac{\mathbf{d}_m^T \mathbf{A} \mathbf{v}_n}{\mathbf{d}_m^T \mathbf{A} \mathbf{d}_m} = -\frac{\mathbf{d}_m^T \mathbf{A} \mathbf{g}_n}{\mathbf{d}_m^T \mathbf{A} \mathbf{d}_m} \qquad (m < n) \quad (2.127)$$

The equation above indicates that $\mathbf{g}_n$ can be written as a linear combination of all previous search directions $\mathbf{d}_0, \cdots, \mathbf{d}_n$:

$$\mathbf{g}_n = \sum_{i=0}^{n} \alpha_i \mathbf{d}_i \quad (2.128)$$

Pre-multiplying $\mathbf{g}_{n+1}^T$ on both sides, we get

$$\mathbf{g}_{n+1}^T \mathbf{g}_k = \mathbf{g}_{n+1}^T \left(\sum_{i=0}^{k} \alpha_i \mathbf{d}_i \right) = \sum_{i=0}^{k} \alpha_i \, \mathbf{g}_{n+1}^T \mathbf{d}_i = 0 \quad (2.129)$$

The last equality is due to $\mathbf{g}_{n+1}^T \mathbf{d}_k = 0$ in Eq. (2.124). We see that $\mathbf{g}_{n+1}$ is also orthogonal to all previous gradients $\mathbf{g}_0, \cdots, \mathbf{g}_n$.

Pre-multiplying $\mathbf{g}_k^T (k \geq n)$ on both sides of Eq. (2.126), we get

$$\mathbf{g}_k^T \mathbf{d}_n = -\mathbf{g}_k^T \mathbf{g}_n - \sum_{m=0}^{n-1} \beta_{mn} \, \mathbf{g}_k^T \mathbf{d}_m = -\mathbf{g}_k^T \mathbf{g}_n = \begin{cases} -||\mathbf{g}_n||^2 & n = k \\ 0 & n < k \end{cases} \quad (2.130)$$

Note that all terms in the summation are zero as $\mathbf{g}_k^T \mathbf{d}_m = 0$ for all $m < n \leq k$ (Eq. (2.124)). Substituting $\mathbf{g}_n^T \mathbf{d}_n = -||\mathbf{g}_n||^2$ into Eq. (2.117), we get

$$\delta_n = -\frac{\mathbf{g}_n^T \mathbf{d}_n}{\mathbf{d}_n^T \mathbf{A} \mathbf{d}_n} = \frac{||\mathbf{g}_n||^2}{\mathbf{d}_n^T \mathbf{A} \mathbf{d}_n} \quad (2.131)$$

Next we consider

$$\mathbf{g}_{m+1} = \mathbf{A}\mathbf{x}_{m+1} - \mathbf{b} = \mathbf{A}(\mathbf{x}_m + \delta_m \mathbf{d}_m) - \mathbf{b} = (\mathbf{A}\mathbf{x}_m - \mathbf{b}) + \delta_m \mathbf{A} \mathbf{d}_m = \mathbf{g}_m + \delta_m \mathbf{A} \mathbf{d}_m \quad (2.132)$$

Pre-multiplying $\mathbf{g}_n^T$ with $n > m$ on both sides we get

$$\mathbf{g}_n^T \mathbf{g}_{m+1} = \mathbf{g}_n^T \mathbf{g}_m + \delta_m \mathbf{g}_n^T \mathbf{A} \mathbf{d}_m = \delta_m \mathbf{g}_n^T \mathbf{A} \mathbf{d}_m \quad (2.133)$$

where $\mathbf{g}_n^T\mathbf{g}_m = 0$ $(m \neq n)$. Solving for $\mathbf{g}_n^T\mathbf{A}\mathbf{d}_m$ we get

$$\mathbf{g}_n^T\mathbf{A}\mathbf{d}_m = \frac{1}{\delta_m}\mathbf{g}_n^T\mathbf{g}_{m+1} = \begin{cases} ||\mathbf{g}_n||^2/\delta_{n-1} & m = n-1 \\ 0 & m < n-1 \end{cases} \tag{2.134}$$

Substituting this into Eq. (2.127) we get

$$\beta_{nm} = -\frac{\mathbf{d}_m^T\mathbf{A}\mathbf{g}_n}{\mathbf{d}_m^T\mathbf{A}\mathbf{d}_m} = \begin{cases} -||\mathbf{g}_n||^2/\delta_{n-1}\mathbf{d}_{n-1}^T\mathbf{A}\mathbf{d}_{n-1} & m = n-1 \\ 0 & m < n-1 \end{cases} \tag{2.135}$$

which is non-zero only when $m = n-1$, i.e., there is only one non-zero term in the summation of the Gram-Schmidt formula for $\mathbf{d}_n$. This is the reason why we choose $\mathbf{v}_n = -\mathbf{g}_n$. We can now drop the second subscript m in β_{nm}, and Eq. (2.125) becomes:

$$\mathbf{d}_n = \mathbf{v}_n - \sum_{m=0}^{n-1}\beta_{nm}\mathbf{d}_m = -\mathbf{g}_n - \beta_n\mathbf{d}_{n-1} \tag{2.136}$$

Substituting the step size $\delta_{n-1} = ||\mathbf{g}_{n-1}||^2/\mathbf{d}_{n-1}^T\mathbf{A}\mathbf{d}_{n-1}$ (Eq. (2.131)) into the above expression for $\beta_{nm} = \beta_m$, we get

$$\beta_n = -\frac{||\mathbf{g}_n||^2}{||\mathbf{g}_{n-1}||^2} \tag{2.137}$$

We note that matrix $\mathbf{A}$ no longer appears in the expression.

The CG algorithm

Summarizing the above, we finally get the conjugate gradient algorithm in the following steps:

1. Set $n = 0$ and initialize the search direction (same as gradient descent):

$$\mathbf{d}_0 = -\mathbf{g}_0 \tag{2.138}$$

2. Terminate if the error $\varepsilon = ||\mathbf{g}_n||^2$ is smaller than a preset threshold. Otherwise, continue with the following:
3. Find optimal step size (Eq. (2.131)) and step forward

$$\delta_n = \frac{||\mathbf{g}_n||^2}{\mathbf{d}_n^T\mathbf{A}\mathbf{d}_n}, \qquad \mathbf{x}_{n+1} = \mathbf{x}_n + \delta_n\mathbf{d}_n \tag{2.139}$$

4. Update gradient:

$$\mathbf{g}_{n+1} = \frac{d}{d\mathbf{x}}f(\mathbf{x}_{n+1}) \tag{2.140}$$

5. Find coefficient for the Gram-Schmidt process (Eq. (2.137):

$$\beta_{n+1} = -\frac{||\mathbf{g}_{n+1}||^2}{||\mathbf{g}_n||^2} \tag{2.141}$$

6. Update search direction (Eq. (2.136)):

$$\mathbf{d}_{n+1} = -\mathbf{g}_{n+1} - \beta_{n+1}\mathbf{d}_n \tag{2.142}$$

Set $n = n + 1$ and go back to step 2.

The algorithm above assumes the objective function $f(\mathbf{x})$ to be quadratic with known $\mathbf{A}$. But when $f(\mathbf{x})$ is not quadratic, $\mathbf{A}$ is no longer available, we can still approximate $f(\mathbf{x})$ as a quadratic function by the first three terms in its Taylor series in the neighborhood of its minimum. Also, the we need to modify the algorithm so that it does not depend on $\mathbf{A}$. Specifically, the optimal step size δ_n calculated in step 3 above based on $\mathbf{A}$ can also be alternatively found by line minimization based on any suitable algorithms for 1-D optimization.

The Matlab code for the conjugate gradient algorithm is listed below:

```
function xn=myCG(j,tol)       % j is the objective function to minimize
    syms d;                   % variable for 1-D symbolic function f(d)
    x=symvar(j).';            % symbolic variables in objective function
    J=matlabFunction(j);      % the objective function
    G=jacobian(j).';          % symbolic gradient of objective function
    G=matlabFunction(G);      % the gradient function
    xn=zeros(length(x),1);    % initial guess of x
    xc=num2cell(xn);
    e=J(xc{:});               % initial error
    gn=G(xc{:});              % gradient at xn
    dn=-gn;                   % use negative gradient as search direction
    n=0;
    while e>tol
        n=n+1;
        f=subs(j,x,xn+d*dn);  % convert n-D f(x) to 1-D f(d)
        delta=Opt1d(f);       % find delta that minimizes 1D function f(d)
        xn=xn+delta*dn;       % updata variable x
        xc=num2cell(xn);
        e=J(xc{:});           % new error
        gn1=G(xc{:});         % new gradient
        bt=-(gn1.'*gn1)/(gn.'*gn);    % find beta
        dn=-gn1-bt*dn;        % new search direction
        gn=gn1;               % update gradient
        fprintf('%d\t(%.4f, %.4f, %.4f)\t%e\n',n,xn(1),xn(2),xn(3),e)
    end
end
```

Here is the function that uses Newton's method to find the optimal step size δ that minimizes the objective function as a 1-D function of δ:

```
function x=Opt1d(f)         % f is 1-D symbolic function to minimize
    syms x;
```

```
    tol=10^(-3);
    d1=diff(f);              % 1st order derivative
    d2=diff(d1);             % 2nd order derivative
    f=matlabFunction(f);
    d1=matlabFunction(d1);
    d2=matlabFunction(d2);
    x=0.0;                   % initial guess of delta
    if d2(x)<=0              % 2nd order derivative needs to be positive
        x=rand-0.5;e
    end
    y=x+1;
    while abs(x-y) > tol     % minimization
        y=x;
        x=y-d1(y)/d2(y);     % Newton iteration
    end
end
```

Example 2.7 To compare the conjugate method and the gradient descent method, consider a very simple 2-D quadratic function

$$f(x,y) = \mathbf{x}^T\mathbf{A}\mathbf{x} = [x_1,\, x_2] \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

The performance of the gradient descent method depends significantly on the initial guess. For the specific initial guess of $\mathbf{x}_0 = [1.5,\ -0.75]^T$, the iteration gets into a zigzag pattern and the convergence is very slow, as shown below:

n	$\mathbf{x} = [x_1,\, x_2]$	$f(\mathbf{x})$
0	$1.500000, -0.750000$	2.812500
1	$0.250000, -0.750000$	$0.468750e-01$
2	$0.250000, -0.125000$	$7.812500e-02$
3	$0.041667, -0.125000$	$1.302083e-02$
4	$0.041667, -0.020833$	$2.170139e-03$
5	$0.006944, -0.020833$	$3.616898e-04$
6	$0.006944, -0.003472$	$6.028164e-05$
7	$0.001157, -0.003472$	$1.004694e-05$
8	$0.001157, -0.000579$	$1.674490e-06$
9	$0.000193, -0.000579$	$2.790816e-07$
10	$0.000193, -0.000096$	$4.651361e-08$
11	$0.000032, -0.000096$	$7.752268e-09$
12	$0.000032, -0.000016$	$1.292045e-09$
13	$0.000005, -0.000016$	$2.153408e-10$

However, as expected, the conjugate gradient method takes exactly $N = 2$ steps from any initial guess to reach at the solution:

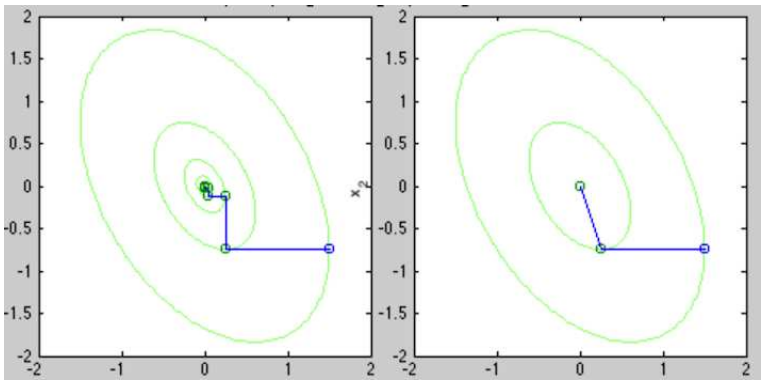


Figure 2.14 Gradient (left) vs. Conjugate Gradient (right)

n	$\mathbf{x} = [x_1, x_2]$	$f(\mathbf{x})$
0	$1.500000, -0.750000$	$2.812500e+00$
1	$0.250000, -0.750000$	$4.687500e-01$
2	$0.000000, -0.000000$	$1.155558e-33$

In Fig. 2.14, the search paths of these two methods are compared.

For an $N = 3$ example of $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}$ with

$$\mathbf{A} = \begin{bmatrix} 5 & 3 & 1 \\ 3 & 4 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

from an initial guess $\mathbf{x}_0 = [1,\ 2,\ 3]^T$, it takes the gradient descent method 41 iterations to reach $\mathbf{x}_{41} = [3.5486e-06,\ -7.4471e-06,\ 4.6180e-06]^T$ corresponding to $f(\mathbf{x}) = 8.5429e-11$. From the same initial guess, it takes the conjugate gradient method only $N = 3$ iterations to converge to the solution:

n	$\mathbf{x} = [x_1, x_2, x_3]$	$f(\mathbf{x})$
0	$1.000000, 2.000000, 3.000000$	$4.500000e+01$
1	$-0.734716, -0.106441, 1.265284$	$2.809225e+00$
2	$0.123437, -0.209498, 0.136074$	$3.584736e-02$
3	$-0.000000, 0.000000, 0.000000$	$3.949119e-31$

For an $N = 9$ example, it takes over 4000 iterations for the gradient descent method to converge with $||\mathbf{e}|| \approx 10^{-10}$, but exactly 9 iterations for the CG method to converge with $||\mathbf{e}|| \approx 10^{-16}$.

Example 2.8 The search path of the conjugate gradient method applied to the minimization of the Rosenbrock function is shown in Fig. 2.15.

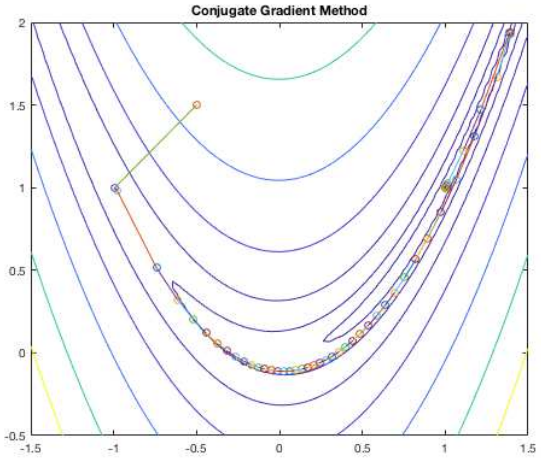


Figure 2.15 Conjugate Gradient Applied to Rosenbrock Surface

Example 2.9 Solve the following $N = 3$ non-linear equation system by the CG method:

$$\begin{cases} f_1(x_1,\, x_2,\, x_3) = 3x_1 - (x_2 x_3)^2 - 3/2 \\ f_2(x_1,\, x_2,\, x_3) = 4x_1^2 - 625\, x_2^2 + 2x_2 - 1 \\ f_3(x_1,\, x_2,\, x_3) = exp(-x_1 x_2) + 20x_3 + 9 \end{cases}$$

The solution is known to be $x_1 = 0.5,\ x_2 = 0,\ x_3 = -0.5$.

This equation system can be represented in vector form as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ and the objective function is $J(\mathbf{x}) = \mathbf{f}^T(\mathbf{x})\mathbf{f}(\mathbf{x})$. The iteration of the CG method with an

initial guess $\mathbf{x}_0 = \mathbf{0}$ is shown below:

n	$\mathbf{x} = [x_1, x_2, x_3]$	$J(\mathbf{x})$
0	$0.0000, 0.0000, 0.0000$	$1.032500e+02$
1	$0.0113, 0.0050, -0.5001$	$3.160163e+00$
2	$0.0188, -0.0021, -0.5004$	$3.095894e+00$
3	$0.5009, -0.0018, -0.5004$	$7.268252e-05$
4	$0.5009, -0.0017, -0.5000$	$1.051537e-05$
5	$0.5008, -0.0012, -0.5000$	$6.511151e-06$
6	$0.5001, -0.0005, -0.5000$	$6.365321e-07$
7	$0.5001, -0.0005, -0.5000$	$5.667357e-07$
8	$0.5002, -0.0004, -0.5000$	$2.675128e-07$
9	$0.5001, -0.0003, -0.5000$	$1.344218e-07$
10	$0.5001, -0.0002, -0.5000$	$1.241196e-07$
11	$0.5000, -0.0001, -0.5000$	$2.120969e-08$
12	$0.5000, -0.0001, -0.5000$	$1.541814e-08$
13	$0.5000, -0.0001, -0.5000$	$7.282025e-09$
14	$0.5000, -0.0001, -0.5000$	$4.801781e-09$
15	$0.5000, -0.0000, -0.5000$	$4.463926e-09$

In comparison, the gradient descent method would need to take over 200 iterations (with much reduced complexity though) to reach this level of error.

Conjugate gradient method used for solving linear equation systems

As discussed before, if $\mathbf{x}$ is the solution that minimizes the quadratic function $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}/2 - \mathbf{b}^T\mathbf{x} + c$, with $\mathbf{A} = \mathbf{A}^T$ being symmetric and positive definite, it also satisfies $d\,f(\mathbf{x})/d\mathbf{x} = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$. In other words, the optimization problem is equivalent to the problem of solving the linear system $\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$, both can be solved by the conjugate gradient method.

Now consider solving the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ with $\mathbf{A} = \mathbf{A}^T$. Let $\mathbf{d}_i$, $(i = 1, \cdots, N)$ be a set of N A-orthogonal vectors satisfying $\mathbf{d}_i^T\mathbf{A}\mathbf{d}_j = 0$ $(i \neq j)$, which can be generated based on any N independent vectors, such as the standard basis vectors, by the Gram-Schmidt method. The solution $\mathbf{x}$ of the equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ can be represented by these N vectors as

$$\mathbf{x} = \sum_{i=1}^{N} c_i \mathbf{d}_i \tag{2.143}$$

Now we have

$$\mathbf{b} = \mathbf{A}\mathbf{x} = \mathbf{A}\left[\sum_{i=1}^{N} c_i \mathbf{d}_i\right] = \sum_{i=1}^{N} c_i \mathbf{A}\mathbf{d}_i \tag{2.144}$$

Pre-multiplying $\mathbf{d}_j^T$ on both sides we get

$$\mathbf{d}_j^T\mathbf{b} = \sum_{i=1}^{N} c_i \mathbf{d}_j^T \mathbf{A}\mathbf{d}_i = c_j \mathbf{d}_j^T \mathbf{A}\mathbf{d}_j \tag{2.145}$$

Solving for c_j we get

$$c_j = \frac{\mathbf{d}_j^T\mathbf{b}}{\mathbf{d}_j^T\mathbf{A}\mathbf{d}_j} \tag{2.146}$$

Substituting this back into the expression for $\mathbf{x}$ we get the solution of the equation:

$$\mathbf{x} = \sum_{i=1}^{N} c_i \mathbf{d}_i = \sum_{i=1}^{N} \left(\frac{\mathbf{d}_i^T\mathbf{b}}{\mathbf{d}_i^T\mathbf{A}\mathbf{d}_i} \right) \mathbf{d}_i \tag{2.147}$$

Also note that as $\mathbf{b} = \mathbf{A}\mathbf{x}$, the ith term of the summation above is simply the A-projection of $\mathbf{x}$ onto the ith direction $\mathbf{d}_i$:

$$\mathbf{p}_{\mathbf{d}_i}(\mathbf{x}) = \left(\frac{\mathbf{d}_i^T\mathbf{A}\mathbf{x}}{\mathbf{d}_i^T\mathbf{A}\mathbf{d}_i} \right) \mathbf{d}_i \tag{2.148}$$

One application of the conjugate gradient method is to solve the normal equation to find the least-square solution of an over-constrained equation system $\mathbf{A}\mathbf{x} = \mathbf{b}$, where the coefficient matrix $\mathbf{A}$ is M by N with rank $R = N < M$. As discussed previously, the normal equation of this system is

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b} \tag{2.149}$$

Here $\mathbf{A}^T\mathbf{A}$ is an N by N symmetric, positive definite matrix. This normal equation can be solved by the conjugate gradient method.

2.7 Issues of Local/Global Minimum

A main concern in optimization problems is whether the solution obtained by a certain algorithm is a local minimum or a global one. If function $f(\mathbf{x})$ to be optimized is quadratic and definite (either positive or negative definite), then any local extremum (either minimum or maximum) is the unique and global extremum. However, if $f(\mathbf{x})$ is not quadratic, then a local minimum found by the algorithm may not be global. In such a case, the *simulated annealing (SA)* algorithm can be used to search for the global minimum, by allowing up-hill moves in the iteration to avoid getting stuck at a local minimum. SA mimics the annealing process in metallurgy, in which the temperature is carefully controlled to decrease slowly so that the material reaches a state with minimum free energy.

The way the SA method avoids getting stuck at a local minimum is to allow the iteration steps to go up-hill. Specifically, the objective function $f(\mathbf{x})$ evaluated at $\mathbf{x}_n$ is treated as the energy at $\mathbf{x}_n$, and an up-hill move corresponding to an energy increase $\Delta E = f(\mathbf{x}_{n+1}) - f(\mathbf{x}_n)$ can be accepted if some probability function

$P(\Delta E, T)$ defined below is greater than a threshold value Th within a certain range comparable with P:

$$P(\Delta E, T) = e^{-\Delta E/T} > \text{Th} \tag{2.150}$$

where T represents the temperature. Obviously smaller ΔE and higher T will cause higher P, it is more probable for a move to be accepted.

During the entire search process, the temperature T gradually reduces so that the probability of accepting an up-hill move also gradually reduces. In the early stage of the search, it is more probable to go up-hill, thereby getting out of a local minimum; but toward the later part of the search process, such a move is less likely to happen, as presumably in this late stage of the search, the energy level is already low, not likely to be at a local minimum with high energy level. Through out the search, the best result corresponding to he lowest energy level is always updated, to be returned to as the final solution at the end of the process.

Problems

Develop Matlab code (or any other language) to implement the algorithms covered in this chapter to solve the equations in the examples in the text. Try at least three different initial guesses to see how they may affect how quickly the iteration converges.

1. Consider the linear over-constrained system in the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a M by N matrix, $\mathbf{x}$ and $\mathbf{b}$ are respectively N-D and M-D vectors. Specifically, let $M = 9$ and $N = 5$, and generate in Matlab by $rand('seed', 1)$, $A = rand(M, N)$ and $b = rand(M, 1)$, and then find the optimal solution of equation system so that the LS error $\varepsilon = ||\mathbf{Ax} - \mathbf{b}||^2$ is minimized.
2. Plot the following single-variable non-linear functions. Make sure the domain of the function in the plot is wide enough to cover at least one root (zero-crossing):

   $$f_1(x) = e^x - 1/x, \qquad f_2(x) = sin(x) - x^3, \qquad f_3(x) = x^3 + x^2 - 26\cos(x)$$

   Then find their roots by solving the corresponding equations by the Newton-Raphson method.
3. Solve the same equations in the previous problem by minimizing the objective function defined as $o(x) = f^2(x)$ by Newton's method. Try at least three different initial guesses and compare results.
4. Solve the following equation systems by the multivariable Newton-Raphson method. Try at least three different initial guesses and compare results in terms of convergence of the iteration.

$$\begin{cases} f_1(\mathbf{x}) = x_1^2 - 2x_1 + x_2^2 - x_3 + 1 = 0 \\ f_2(\mathbf{x}) = x_1 x_2^2 - x_1 - 3x_2 + x_2 x_3 + 2 = 0 \\ f_3(\mathbf{x}) = x_1 x_3^3 - 3x_3 + x_2 x_3^2 + x_1 x_2 = 0 \end{cases} \qquad \begin{cases} f_1(\mathbf{x}) = 3x - 1 - (x_2 x_3)^2 - 3/2 = 0 \\ f_2(\mathbf{x}) = 4x_1^2 - 500x_2^2 + 2x_2 - 1 = 0 \\ f_3(\mathbf{x}) = exp(-x_1 x_2) + 20x_3 + 9 = 0 \end{cases}$$

5. Resolve the equation systems in the previous problem in vector form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, by minimizing the objective function defined as $o(\mathbf{x}) = ||\mathbf{f}(\mathbf{x})||^2) = \mathbf{f}^T\mathbf{f}$, using Newton's method for minimizing multivariable functions. Try at least three different initial guesses and compare results.
6. Make a 3-D surface plot and a 2-D contour plot of the Rosenbrock function

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

   Use both the gradient descent method (without second order derivatives) and Newton's method to find the minimum of the function from different initial guesses such as: $[-1, 2]$, $[-1, 0]$, $[0, 1]$, $[2, 2]$. Plot the search paths, same as in Fig. 2.7
7. Resolve the same problem above but using various versions of quasi-Newton's method (SR1, BFGS, DFP). Try the same initial guesses and plot the search paths.
8. Resolve the same problem above but using the conjugate gradient method. Try the same initial guesses and plot the search paths.
9. Solve the following equation system by the conjugate gradient method:

$$\begin{cases} f_1(x_1, x_2, x_3) = 3x_1 - (x_2x_3)^2 - 3/2 \\ f_2(x_1, x_2, x_3) = 4x_1^2 - 625\,x_2^2 + 2x_2 - 1 \\ f_3(x_1, x_2, x_3) = exp(-x_1x_2) + 20x_3 + 9 \end{cases}$$

3 Constrained Optimization

This chapter for constrained optimization contains some theoretically intensive discussions about constrained optimization, including theories and methods for optimization with both equality and inequality constraints, and the KKT conditions for both primal and dual problems of constrained optimization. Such discussions are necessary for the study of certain topics in future chapters, such as the method of support vector machines, one of the most important classification method in machine learning, to be considered in Chapter 14.

An optimization problem becomes more complicated if it is constrained, i.e., the arguments $\mathbf{x}$ of the objective function $f(\mathbf{x})$ are subject to (abbreviated as s.t. in the following) certain equality and/or inequality constraints in terms of what values they can take. Such a generic constrained optimization problem can be formulated as:

$$\begin{cases} \text{max/min:} & f(\mathbf{x}) = f(x_1, \cdots, x_N) \\ \text{s. t.:} & \begin{cases} h_i(\mathbf{x}) = 0, & (i = 1, \cdots m) \\ g_j(\mathbf{x}) \leq 0, & (j = 1, \cdots n) \end{cases} \end{cases} \tag{3.1}$$

The set composed of all values of $\mathbf{x}$ that satisfy the $m + n$ constraints is called the *feasible region* of the problem. The goal is to find a point $\mathbf{x}^*$ in the feasible region at which $f(\mathbf{x}^*)$ is an extremum.

In the following we will first consider the most general case where the objective function $f(\mathbf{x})$ is nonlinear and so are the constraint functions $g_i(\mathbf{x})$ and $h_j(\mathbf{x})$. The processing of solving such a problem is called the *nonlinear programming (NLP)*. Here *programming* means the mathematical procedure for solving the optimization problems. We will consider the general method of Lagrange multiplier and a set of conditions all solutions of such problems must satisfy.

We will then consider some specific algorithms for two special cases: *linear programming (LP)* in Section 3.4 for problems where the objective function and all constraint functions are linear (the feasible region is a polytope), and then *quadratic programming (QP)* in Section 3.6 for problems where the objective function is quadratic while the constraints are linear.

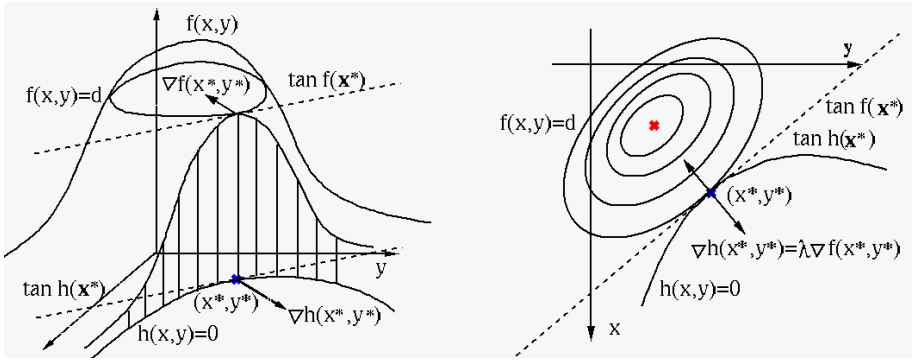


Figure 3.1 Equality Constrained Optimization of Function $f(x, y)$

3.1 Optimization with Equality Constraints

The optimization problems subject to equality constraints can be generally formulated as:

$$\begin{cases} \text{max/min:} & f(\mathbf{x}) = f(x_1, \cdots, x_N) \\ \text{s. t.:} & h_i(\mathbf{x}) = h_i(x_1, \cdots, x_N) = 0, \quad (i = 1, \cdots m) \end{cases} \tag{3.2}$$

Here we assume m is no bigger than N, as in general there does not exist a solution that satisfies more than N equations in the N-D space.

This problem can be visualized in the special case with $N = 2$ and $m = 1$, where both $f(\mathbf{x})$ and $h(\mathbf{x})$ are surfaces defined over the 2-D space spanned by x_1 and x_2, and $h(\mathbf{x}) = 0$ is the intersection line of $h(\mathbf{x})$ and the 2-D plane. The solution $\mathbf{x}^*$ must satisfy the following two conditions:

- $\mathbf{x}^*$ is on the intersection line satisfying $h(\mathbf{x}^*) = 0$;
- $\mathbf{x}^*$ is on the contour line $f(\mathbf{x}) = d$ (for some d as shown in Fig. 3.1, so that $f(\mathbf{x}^*)$ will neither increase nor decrease while moving along the contour line in the neighborhood of $\mathbf{x}^*$.

For both conditions to be satisfied, the two contours $h(\mathbf{x}) = 0$ and $f(\mathbf{x}) = d$ must coincide at $\mathbf{x}^*$, i.e., they must have the same tangent line and therefore parallel gradients (perpendicular to the tangents):

$$\bigtriangledown_{\mathbf{x}} f(\mathbf{x}^*) = \lambda \bigtriangledown_{\mathbf{x}} h(\mathbf{x}^*) \tag{3.3}$$

where $\bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = [\partial f/\partial x_1, \cdots, \partial f/\partial x_N]^T$ is the gradient of $f(\mathbf{x})$, and λ is a constant scaling factor which can be either positive if the two gradients are in the same direction or negative if they are in opposite directions. As here we are only interested in whether the two curves coincide or not, the sign of λ can be either positive or negative.

To find such a point $\mathbf{x}^*$, we first construct a new objective function, called the

Lagrange function (or simply *Lagrangian*):

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda\, h(\mathbf{x}) \tag{3.4}$$

where λ is the *Lagrange multiplier*, which can be either positive or negative (i.e., the sign in front of λ can be either plus or minus), and then set the gradient of this objective function with respect to λ as well as $\mathbf{x}$ to zero:

$$\bigtriangledown_{\mathbf{x},\lambda} L(\mathbf{x}, \lambda) = \bigtriangledown_{\mathbf{x},\lambda} [f(\mathbf{x}) - \lambda\, h(\mathbf{x})] = \mathbf{0} \tag{3.5}$$

This equation contains two parts:

$$\begin{cases} \bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = \lambda \bigtriangledown_{\mathbf{x}} h(\mathbf{x}) \\ \bigtriangledown_{\lambda} L(\mathbf{x}, \lambda) = \partial L(\mathbf{x}, \lambda)/\partial\lambda = -h(\mathbf{x}) = 0 \end{cases} \tag{3.6}$$

by which both Eq. (3.3) and the equality constraint are satisfied as desired.

This is an equation system containing $N + m = 2 + 1 = 3$ equations for the same number of unknowns x_1, x_2, λ:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lambda \frac{\partial h(\mathbf{x})}{\partial x_i} \quad (i = 1, 2), \qquad h(\mathbf{x}) = 0 \tag{3.7}$$

The solution $\mathbf{x}^* = [x_1^*, x_2^*]^T$ and λ^* of this equation system happens to satisfy the two desired relationships: (a) $\bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = \lambda \bigtriangledown_{\mathbf{x}} h(\mathbf{x})$, and (b) $h(\mathbf{x}^*) = 0$.

Specially, consider the following two possible cases:

- If $\lambda^* = 0$, then

$$\bigtriangledown_{\mathbf{x}} f(\mathbf{x}^*) = \lambda^* \bigtriangledown_{\mathbf{x}} h(\mathbf{x}^*) = 0 \tag{3.8}$$

  indicating $f(\mathbf{x}^*)$ is an extremum independent of the constraint, i.e., the constraint $h(\mathbf{x}) = 0$ is *inactive*, and the optimization is unconstrained.
- If $\lambda^* \neq 0$, then $\bigtriangledown_{\mathbf{x}} f(\mathbf{x}^*) \neq 0$ (as in general $\bigtriangledown_{\mathbf{x}} h(\mathbf{x}^*) \neq \mathbf{0}$), indicating $f(\mathbf{x}^*)$ is not an extremum without the constraint, i.e., the constraint is *active*, and the optimization is indeed constrained.

The discussion above can be generalized from 2-D to any $N > 2$ dimensional space, in which the optimal solution $\mathbf{x}^*$ is to be found to extremize the objective $f(\mathbf{x})$ subject to m equality constraints $h_i(\mathbf{x}) = 0$ $(i = 1, \cdots, m)$, each representing a contour surface in the N-D space. The solution $\mathbf{x}^*$ must satisfy the following two conditions:

- $\mathbf{x}^*$ is on all m contour surfaces to satisfy $h_1(\mathbf{x}^*) = \cdots = h_m(\mathbf{x}^*) = 0$, i.e., it is on the intersection of these surfaces, a curve in the space (e.g., the intersection of two surfaces in 3-D space is a curve);
- $\mathbf{x}^*$ is on a contour surface $f(\mathbf{x}^*) = d$ (for some value d) so that $f(\mathbf{x}^*)$ is an extremum, i.e., it does not increase or decrease while moving along the curve in the neighborhood of $\mathbf{x}^*$.

Combining these conditions, we see that the intersection (a curve or a surface) of the m surfaces $h_i(\mathbf{x}) = 0$ must coincide with the contour surface $f(\mathbf{x}) = d$

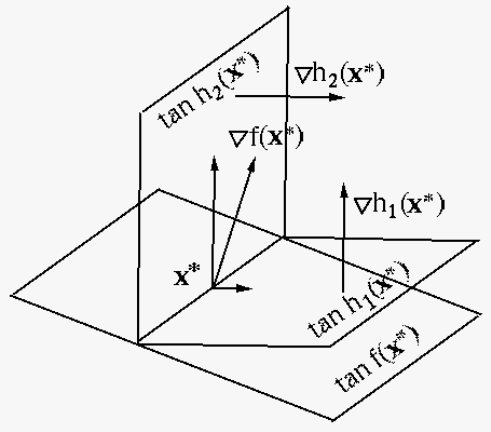


Figure 3.2 Derivation of Lagrange Function

at $\mathbf{x}^*$, i.e., the intersection of the tangent surfaces of $h_i(\mathbf{x}) = 0$ through $\mathbf{x}^*$ must coincide with the tangent surface of $f(\mathbf{x}) = d$ at $\mathbf{x}^*$, therefore the gradient $\bigtriangledown f(\mathbf{x}^*)$ must be a linear combination of the m gradients $\bigtriangledown h_i(\mathbf{x}^*)$:

$$\bigtriangledown_{\mathbf{x}} f(\mathbf{x}^*) = \sum_{i=1}^{m} \lambda_i^* \bigtriangledown_{\mathbf{x}} h_i(\mathbf{x}^*) \tag{3.9}$$

as shown in Fig. 3.2.

To find such a solution $\mathbf{x}^*$ of the optimization problem, we first construct the Lagrange function:

$$L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i=1}^{m} \lambda_i h_i(\mathbf{x}) \tag{3.10}$$

where $\boldsymbol{\lambda} = [\lambda_1, \cdots, \lambda_m]^T$ is a vector for m Lagrange multipliers. We then set its gradient to zero:

$$\bigtriangledown_{\mathbf{x},\boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\lambda}) = \bigtriangledown_{\mathbf{x},\boldsymbol{\lambda}} \left[f(\mathbf{x}) - \sum_{i=1}^{m} \lambda_i h_i(\mathbf{x}) \right] = \mathbf{0} \tag{3.11}$$

to get the following two equation systems of N and m equations respectively:

$$\bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = \sum_{i=1}^{m} \lambda_i \bigtriangledown_{\mathbf{x}} h_i(\mathbf{x}) \quad \text{i.e.} \quad \frac{\partial f(\mathbf{x})}{\partial x_j} = \sum_{i=1}^{m} \lambda_i \frac{\partial h_i(\mathbf{x})}{\partial x_j} \quad (j = 1, \cdots, N), \tag{3.12}$$

and

$$\frac{\partial L(\mathbf{x}, \boldsymbol{\lambda})}{\partial \lambda_i} = h_i(\mathbf{x}) = 0 \quad (i = 1, \cdots, m) \tag{3.13}$$

The first set of N equations indicates that the gradient $\bigtriangledown_{\mathbf{x}} f(\mathbf{x}^*)$ at $\mathbf{x}^*$ is a linear combination of the m gradients $\bigtriangledown_{\mathbf{x}} h_i(\mathbf{x}^*)$, while the second set of m equations

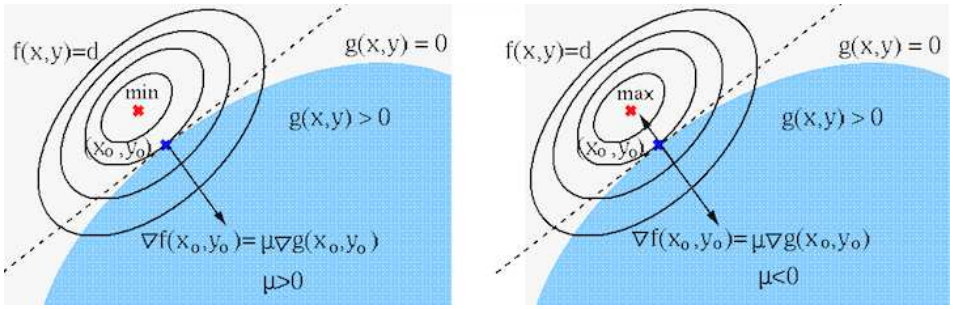


Figure 3.3 Inequality Constrained Optimization of $f(x, y)$

guarantees that $\mathbf{x}^*$ also satisfy the m equality constraints, both as desired. Solving these equations we get the desired solution $\mathbf{x}^*$ together with the Lagrange multipliers $\lambda_1, \cdots, \lambda_m$. This is the *Lagrange multiplier method*, by which the constrained optimization problem is reformulated in such a way that the solution can be found by solving the equation system obtained by setting the gradient of the Lagrange function to zero.

Specially, if $\lambda_i = 0$, then the ith constraint is not active. More specially if $\lambda_i = 0$ for all $i = 1, \cdots, m$, then

$$\bigtriangledown_{\mathbf{x}} f(\mathbf{x}^*) = \sum_{i=1}^{m} \lambda_i \bigtriangledown_{\mathbf{x}} h_i(\mathbf{x}) = \mathbf{0} \tag{3.14}$$

indicating $f(\mathbf{x}^*)$ is an extremum independent of any of the constraints, i.e., the optimization problem is unconstrained.

3.2 Optimization with Inequality Constraints

The optimization problems subject to inequality constraints can be generally formulated as:

$$\begin{cases} \text{max/min:} & f(\mathbf{x}) = f(x_1, \cdots, x_N) \\ \text{s. t.:} & g_j(\mathbf{x}) = g_j(x_1, \cdots, x_N) \leq \text{ or } \geq 0, \qquad (j = 1, \cdots n) \end{cases} \tag{3.15}$$

Again, to visualize the problem we first consider an example with $N = 2$ and $n = 1$, as shown in Fig. 3.3 for the minimization (left) and maximization (right) of $f(\mathbf{x})$ subject to $g(\mathbf{x}) > 0$. Here the constrained solution $\mathbf{x}^* = [x_1^*, x_2^*]^T$ is on the boundary of the feasible region satisfying $g(x_1, x_2) = 0$, while the unconstrained extremum is outside the feasible region.

We consider the following two possible cases.

- First, if the unconstrained extremum is inside the feasible region, i.e., the

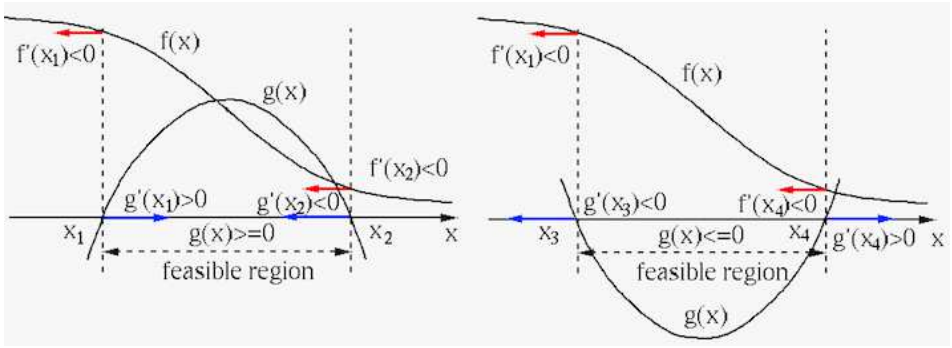


Figure 3.4 Determination of Signs of Lagrange Multipliers

inequality constraint is inactive, then the solution $\mathbf{x}^*$ is the same as that of an unconstrained problem, not on the boundary of the feasible region, i.e.,

$$\nabla_{\mathbf{x}} f(\mathbf{x}^*) = \mathbf{0}, \qquad g(\mathbf{x}^*) \neq 0 \tag{3.16}$$

Similar to the case of an inactive equality constraint, here the multiplier for an inactive inequality constraint is zero, i.e., $\mu^* = 0$, and we get

$$\nabla_{\mathbf{x}} f(\mathbf{x}^*) = \mu^* \ \nabla_{\mathbf{x}} g(\mathbf{x}^*) = \mathbf{0} \tag{3.17}$$

the same as in Eq. (3.8).

- Second, if the unconstrained extremum at which $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$ is outside the feasible region, i.e., the inequality constraint is active, then the constrained solution $\mathbf{x}^*$ must be on the boundary of the feasible region, and it is different from the unconstrained solution:

$$g(\mathbf{x}^*) = 0, \qquad \nabla_{\mathbf{x}} f(\mathbf{x}^*) \neq \mathbf{0} \tag{3.18}$$

In this case, the problem becomes the same as an equality constrained problem considered before, i.e., the objective function $f(\mathbf{x})$ and the constraint function $g(\mathbf{x})$ have the same tangent at $\mathbf{x}^*$ with parallel gradients:

$$\nabla_{\mathbf{x}} f(\mathbf{x}^*) = \mu^* \ \nabla_{\mathbf{x}} g(\mathbf{x}^*) \quad \text{or} \quad \nabla_{\mathbf{x}} f(\mathbf{x}^*) - \mu^* \ \nabla_{\mathbf{x}} g(\mathbf{x}^*) = 0 \tag{3.19}$$

However, different from before, now we also need to consider whether the two gradients have the same or opposite directions, corresponding to $\mu > 0$ or $\mu < 0$, respectively. Specifically, there exist four possible cases in terms of the sign of μ, depending on whether $f(\mathbf{x})$ is to be maximized or minimized, and whether the constraint is $g(\mathbf{x}) \geq 0$ or $g(\mathbf{x}) \leq 0$, as the four points x_1 through x_4 illustrated in the 1-D examples in Fig. 3.4 and also summarized in the tables below:

	$g(\mathbf{x}) \geq 0$	$g(\mathbf{x}) \leq 0$
max $f(\mathbf{x})$	(1) $\bigtriangledown f(\mathbf{x}_1) = \mu \bigtriangledown g(\mathbf{x}_1),\ \mu < 0$	(3) $\bigtriangledown f(\mathbf{x}_3) = \mu \bigtriangledown g(\mathbf{x}_3),\ \mu > 0$
min $f(\mathbf{x})$	(2) $\bigtriangledown f(\mathbf{x}_2) = \mu \bigtriangledown g(\mathbf{x}_2),\ \mu > 0$	(4) $\bigtriangledown f(\mathbf{x}_4) = \mu \bigtriangledown g(\mathbf{x}_4),\ \mu < 0$

(3.20)

	max/min	subject to	μ
x_1	max $f(\mathbf{x})$	$g(\mathbf{x}) \geq 0$	$\mu < 0$
x_2	min $f(\mathbf{x})$	$g(\mathbf{x}) \geq 0$	$\mu > 0$
x_3	max $f(\mathbf{x})$	$g(\mathbf{x}) \leq 0$	$\mu > 0$
x_4	min $f(\mathbf{x})$	$g(\mathbf{x}) \leq 0$	$\mu < 0$

(3.21)

We can also consider the product of μ and $g(\mathbf{x})$ for either maximization or minimization:

$$\mu\, g(\mathbf{x}) \begin{cases} \leq 0 & \text{for maximization} \\ \geq 0 & \text{for minimization} \end{cases} \tag{3.22}$$

Summarizing the two cases above, we see that $g(\mathbf{x}^*) \neq 0$ but $\mu^* = 0$ in the first case of an inactive constraint, and $g(\mathbf{x}^*) = 0$ but $\mu^* \neq 0$ in the second case of an active constraint, i.e., the following holds in both cases:

$$\mu^*\, g(\mathbf{x}^*) = 0 \tag{3.23}$$

The discussion above can be generalized from 2-D to $N > 2$ dimensional space, in which the optimal solution $\mathbf{x}^*$ is to be found to extremize the objective $f(\mathbf{x})$ subject to n inequality constraints $g_i(\mathbf{x}) = 0$ $(i = 1, \cdots, n)$. To solve this inequality constrained optimization problem, we first construct the Lagrangian:

$$L(\mathbf{x}, \boldsymbol{\mu}) = f(\mathbf{x}) - \sum_{i=1}^{n} \mu_i\, g_i(\mathbf{x}) \tag{3.24}$$

where $\boldsymbol{\mu} = [\mu_1, \cdots, \mu_n]^T$ is a vector for n Lagrange multipliers. (We note that in some literature, a plus sign is used in front of the summation of the second term. This is equivalent to our discussion here so long as the sign of μ indicated in Table 3.21 is negated.) Now Eq. (3.23) can be written as

$$(\boldsymbol{\mu}^*)^T\, g(\mathbf{x}^*) = 0 \tag{3.25}$$

We then set the gradient of the Lagrangian to zero:

$$\bigtriangledown_{\mathbf{x},\boldsymbol{\mu}} L(\mathbf{x}, \boldsymbol{\mu}) = \bigtriangledown_{\mathbf{x},\boldsymbol{\mu}} \left[f(\mathbf{x}) - \sum_{i=1}^{n} \mu_i\, g_i(\mathbf{x}) \right] = \mathbf{0} \tag{3.26}$$

and get two equation systems of N and n equations, respectively:

$$\bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = \sum_{i=1}^{n} \mu_i \bigtriangledown_{\mathbf{x}} g_i(\mathbf{x}) \tag{3.27}$$

and

$$\bigtriangledown_{\boldsymbol{\mu}} \left[\sum_{i=1}^{n} \mu_i \, g_i(\mathbf{x}) \right] = \mathbf{0} \quad \text{i.e.} \quad \frac{\partial L(\mathbf{x}, \boldsymbol{\mu})}{\partial \mu_i} = g_i(\mathbf{x}) = 0 \quad (i = 1, \cdots, n) \tag{3.28}$$

The result above for the inequality constrained problems based on Eq. (3.26) looks the same as that for the equality constrained problems based on Eq. (3.11), but with an additional requirement regarding the sign of the Lagrange multipliers. For an equality constrained problem, the direction of the gradient $\bigtriangledown g(\mathbf{x})$ is of no concern, i.e., the sign of λ is unrestricted; but here for an inequality constrained problem, the sign of μ needs to be consistent with those shown in Table 3.21, unless when the constraint is inactive and $\mu = 0$.

As a simple example, consider the minimization of a function $f(\mathbf{x})$ by the method of gradient descent. To find the best increment $\Delta\mathbf{x}$ that minimizes $f(\mathbf{x}+\Delta\mathbf{x}) \approx f(\mathbf{x}) + \mathbf{g}_f(\mathbf{x})^T \Delta\mathbf{x}$ subjected to the constraint of a limited length, i.e., $||\Delta\mathbf{x}||^2/2 \leq C$, we first construct the Lagrangian:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \mathbf{g}_f(\mathbf{x})^T \Delta\mathbf{x} - \mu(||\Delta\mathbf{x}||^2/2 - C) \tag{3.29}$$

and set its derivative to zero:

$$\frac{d}{d\Delta\mathbf{x}} L(\mathbf{x}, \lambda)\mathbf{g}_f(\mathbf{x}) - \mu\Delta\mathbf{x} = \mathbf{0} \tag{3.30}$$

where $\mu < 0$ according to Table 3.21. We then solve the equation to get

$$\Delta\mathbf{x} = \frac{1}{\mu}\mathbf{g}_f(\mathbf{x}) \tag{3.31}$$

and the iteration

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \delta\mathbf{g}(\mathbf{x}_n) \tag{3.32}$$

same as the gradient descent method in Eq. (2.25).

We now consider the general optimization of an N-D objective function $f(\mathbf{x})$ subject to multiple constraints of qualities and inequalities:

$$\begin{array}{ll} \text{max/min:} & f(\mathbf{x}) = f(x_1, \cdots, x_N) \\ \text{s. t.:} & \begin{cases} h_i(\mathbf{x}) = 0, & (i = 1, \cdots m) \\ g_j(\mathbf{x}) \leq 0 \ \text{ or } \ g_j(\mathbf{x}) \geq 0, & (j = 1, \cdots n) \end{cases} \end{array} \tag{3.33}$$

For notational convenience, we denote the $m + n$ equality and inequality constraints in vector form as:

$$\begin{array}{ll} \text{max/min:} & f(\mathbf{x}) \\ \text{s. t.:} & \begin{cases} \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \ \text{ or } \ \mathbf{g}(\mathbf{x}) \geq \mathbf{0} \end{cases} \end{array} \tag{3.34}$$

where $\mathbf{h}(\mathbf{x}) = [h_1(\mathbf{x}), \cdots, h_m(\mathbf{x})]^T$ and $\mathbf{g}(\mathbf{x}) = [g_1(\mathbf{x}), \cdots, g_n(\mathbf{x})]^T$.

To solve this optimization problem, we first construct the Lagrangian

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) - \sum_{i=1}^{m} \lambda_i h_i(\mathbf{x}) - \sum_{j=1}^{n} \mu_j g_j(\mathbf{x}) = f(\mathbf{x}) - \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{x}) - \boldsymbol{\mu}^T \mathbf{g}(\mathbf{x}) \tag{3.35}$$

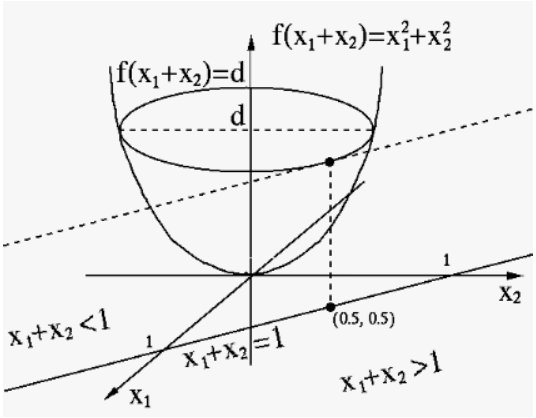


Figure 3.5 Inequality Constrained Optimization

where the Lagrange multipliers in $\boldsymbol{\lambda} = [\lambda_1, \cdots, \lambda_m]^T$ and $\boldsymbol{\mu} = [\mu_1, \cdots, \mu_n]^T$ are for the m equality and n non-negative constraints, respectively. We then set to zero the gradient of $L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ with respect to both $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$ as well as $\mathbf{x}$.

$$\frac{\partial}{\partial \mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}, \quad \frac{\partial}{\partial \boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}, \quad \frac{\partial}{\partial \boldsymbol{\mu}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0} \tag{3.36}$$

Solving these equation systems we get the solution $\mathbf{x}^*$. While λ_i can be either positive or negative, the sign of μ_j needs to be consistent with those specified in Table 3.21. Otherwise the inequality constraints is inactive.

Example 3.1 Find the extremum of $f(x_1, x_2) = x_1^2 + x_2^2$ subject to each of the three different constraints: $x_1 + x_2 = 1$, $x_1 + x_2 \leq 1$, and $x_1 + x_2 \geq 1$, as illustrated in Fig. 3.5.

- $$\begin{cases} \text{min:} & f(x_1, x_2) = x_1^2 + x_2^2 \\ \text{s. t.:} & h(x_1, x_2) = x_1 + x_2 - 1 = 0 \end{cases}$$

  The Lagrangian is:

  $$L(x_1, x_2, \lambda) = f(x_1, x_2) - \lambda g(x_1, x_2) = x_1^2 + x_2^2 - \lambda(x_1 + x_2 - 1)$$

  Solving the following equations

  $$\frac{\partial L}{\partial x_1} = 2x_1 - \lambda = 0, \quad \frac{\partial L}{\partial x_2} = 2x_2 - \lambda = 0, \quad \frac{\partial L}{\partial \lambda} = x_1 + x_2 - 1 = 0$$

  we get $\lambda^* = 1,\ x_1^* = x_2^* = 0.5$, i.e., the function is minimized at $f(0.5,\ 0.5) = 0.5$. We note that $f(x_1, x_2)$ and $g(x_1, x_2)$ have the same gradients at $x_1^* = x_2^* = 0.5$:

  $$\bigtriangledown f(0.5, 0.5) = \bigtriangledown h(0.5, 0.5) = [1, 1]^T$$

- Consider these two problems:

$$\begin{cases} \text{min:} & f(x_1, x_2) = x_1^2 + x_2^2 \\ \text{s. t.:} & g(x_1, x_2) = x_1 + x_2 - 1 \geq 0 \end{cases} \qquad \begin{cases} \text{max:} & f(x_1, x_2) = x_1^2 + x_2^2 \\ \text{s. t.:} & g(x_1, x_2) = x_1 + x_2 - 1 \leq 0 \end{cases}$$

  These problems are the two middle cases (2 and 3) in Table 3.21. They have the same Lagrangian as above with the same results $x_1^* = x_2^* = 0.5$ and $\mu^* = 1 > 0$. According to Table 3.21, $\mathbf{x}^*$ is the solution for the minimization problem subject to $g(\mathbf{x}) \geq 0$ (left), or the maximization problem subject to $g(\mathbf{x}) \leq 0$.
- Consider these two problems:

$$\begin{cases} \text{max:} & f(x_1, x_2) = x_1^2 + x_2^2 \\ \text{s. t.:} & g(x_1, x_2) = x_1 + x_2 - 1 \geq 0 \end{cases} \qquad \begin{cases} \text{min:} & f(x_1, x_2) = x_1^2 + x_2^2 \\ \text{s. t.:} & g(x_1, x_2) = x_1 + x_2 - 1 \leq 0 \end{cases}$$

  These problems are cases 1 and 4 in Table 3.21. Again they have the same Lagrangian as in the previous cases and the same results $x_1^* = x_2^* = 0.5$, and $\mu^* = 1 > 0$. According to Table 3.21, $\mathbf{x}^*$ is *not* the solution of either of the two problems, i.e., the constraint is inactive. We therefore need to assume $\mu^* = 0$ and solve the following equations for an unconstrained problem:

$$\frac{\partial L}{\partial x_1} = 2x_1 = 0, \quad \frac{\partial L}{\partial x_2} = 2x_2 = 0$$

  Now we get $x_1^* = x_2^* = 0$ and $\mathbf{x}^* = \mathbf{0}$, with minimum $f(\mathbf{x}^*) = 0$. This is the solution for the minimization problem (right), at which $g(\mathbf{x}^*) \neq 0$, i.e., the constraint is inactive. However this is not the solution for the maximization problem as the function is unbounded from above.

3.3 Duality and KKT Conditions

Previously we considered the constrained minimization problem:

$$\begin{cases} \text{min:} & f_p(\mathbf{x}) \\ \text{s. t.:} & \mathbf{h}(\mathbf{x}) = \mathbf{0}, \qquad \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \quad \text{or} \quad \mathbf{g}(\mathbf{x}) \geq \mathbf{0} \end{cases} \tag{3.37}$$

with its Lagrangian:

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f_p(\mathbf{x}) - \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{x}) - \boldsymbol{\mu}^T \mathbf{g}(\mathbf{x}) \tag{3.38}$$

The optimal solution $\mathbf{x}^*$ can be found by solving the following equation systems:

$$\nabla_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}, \quad \nabla_{\boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}, \quad \nabla_{\boldsymbol{\mu}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0} \tag{3.39}$$

at which the objective function reaches its minimum $p^* = f_p(\mathbf{x}^*)$.

In particular, if the objective function $f_p(\mathbf{x})$ and all inequality constraints $\mathbf{g}_i(\mathbf{x})$ are convex and the equality constraints $\mathbf{h}_i(\mathbf{x})$ are linear (affine), i.e., the feasible region is convex (Section A.8.3), then the problem is a *convex optimization* with some nice properties as shown below.

This constrained minimization problem given above can be called the *primal problem*, as there also exists an equivalent *dual problem*, which may be easier to solve. To construct the dual problem, we first define the *dual function* as the lower bound of the Lagrangian in Eq. (3.38) to be minimized:

$$f_d(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \min_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \min_{\mathbf{x}} \left[f_p(\mathbf{x}) - \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{x}) - \boldsymbol{\mu}^T \mathbf{g}(\mathbf{x}) \right] \tag{3.40}$$

We further find the tightest lower bound, the maximum of the dual function $f_d(\boldsymbol{\lambda}, \boldsymbol{\mu})$ over its variables $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$, by solving the following dual problem:

$$\begin{cases} \text{max:} & f_d(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \min_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \\ \text{s. t.:} & \boldsymbol{\mu} \leq \mathbf{0} \quad \text{or} \quad \boldsymbol{\mu} \geq \mathbf{0} \end{cases} \tag{3.41}$$

Here $\boldsymbol{\lambda}$ for the equality constrains of the primal problem can be either positive or negative, i.e., it is not constrained in the dual problem. However, the sign of $\boldsymbol{\mu}$ for the inequality constraints of the primal minimization problem needs to satisfy the following according to Eq. (3.22) (the second inequality for this minimization primal problem):

$$\boldsymbol{\mu}^T \mathbf{g}(\mathbf{x}) \geq 0 \tag{3.42}$$

i.e., $\boldsymbol{\mu}$ is subject to the inequality constraint of the dual problem. We denote the optimal solution of the dual problem by $\{\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*\}$, and the maximum of the dual objective function by $d^* = f_d(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$.

Obviously the primal objective function of any feasible $\mathbf{x}$ is always no less than its lower bound, the dual objective function of any feasible $\{\boldsymbol{\lambda}, \boldsymbol{\mu}\}$:

$$f_p(\mathbf{x}) \geq f_d(\boldsymbol{\lambda}, \boldsymbol{\mu}) \tag{3.43}$$

and this inequality also holds for the optimal solutions $\mathbf{x}^*$ and $\{\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*\}$:

$$p^* = f_p(\mathbf{x}) \geq f_d(\boldsymbol{\lambda}, \boldsymbol{\mu}) = d^* \tag{3.44}$$

The difference $p^* - d^*$ is defined as the *duality gap*. Obviously $p^* - d^* \geq 0$ always holds and the duality is called *weak duality*. In some problems where certain conditions (*Slater's conditions*) are satisfied, $p^* = d^*$, i.e., d^* is the tightest lower bound, the reality is called *strong duality*.

We now consider the necessary and sufficient conditions for the solutions of both the primal and dual problems. We first introduce the *Karush-Kuhn-Tucker (KKT)* conditions:

- *Stationarity:* $\mathbf{x}^*$ maximizes or minimizes the Lagrangian, i.e.,

$$\nabla_{\mathbf{x}} L(\mathbf{x}^*, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0} \tag{3.45}$$

- *Primal feasibility:* $\mathbf{x}^*$ must be inside the feasible region:

$$h_i(\mathbf{x}^*) = 0, \quad g_j(\mathbf{x}^*) \leq 0 \quad \text{or} \quad g_j(\mathbf{x}^*) \geq 0, \qquad (i = 1, \cdots, m,\ j = 1, \cdots, n) \tag{3.46}$$

- *Dual feasibility:* All multipliers in $\boldsymbol{\mu}$ for the inequalities must take proper sign as specified by Table 3.21:

$$\mu_j^* \leq 0 \quad \text{or} \quad \mu_j^* \geq 0, \quad (j = 1, \cdots, n) \tag{3.47}$$

- *Complementarity:* Either μ_j^* or $g_j(\mathbf{x}^*)$ must be zero depending on whether the jth constraint is inactive or not:
  - if inactive, then $g_j(\mathbf{x}^*) \neq 0$, but $\mu_j^* = 0$;
  - if active, then $\mu_j^* \neq 0$, but $g_j(\mathbf{x}^*) = 0$.

  Combining both cases, we get (Eq. (3.25)):

$$(\boldsymbol{\mu}^*)^T \mathbf{g}(\mathbf{x}^*) = 0 \tag{3.48}$$

We now prove the following theorem stating that the KKT conditions and both necessary and sufficient for the solutions of the primal and dual problems to exist.

Theorem:

- (Necessary) If $\mathbf{x}^*$ and $\{\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*\}$ are solutions for the respective primal and dual problems with strong duality, then they satisfy all KKT conditions.
- (Sufficient) If $\mathbf{x}^*$ and $\{\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*\}$ satisfy all KKT conditions, then they are solutions for the respective primal and dual problems, and strong duality holds, when the primal objective function $f_p(\mathbf{x}^*)$ and the inequality constraints $g_j(\mathbf{x})$ are convex, and equality constraints $h_j(\mathbf{x})$ are linear.

Proof:

- Necessary conditions:

  As $\mathbf{x}^*$ and $\{\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*\}$ are respectively the primal and dual solutions, the primal and dual feasibilities must be satisfied, and so is stationarity as the optimal solution $\mathbf{x}^*$ satisfies Eq. (3.39). In addition, we also have $\mathbf{h}(\mathbf{x}^*) = \mathbf{0}$. To show complementarity also holds, consider

$$\begin{aligned} d^* &= f_d(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = \min_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \\ &= \min_{\mathbf{x}} \left[f_p(\mathbf{x}) - (\boldsymbol{\lambda}^*)^T \mathbf{h}(\mathbf{x}) - (\boldsymbol{\mu}^*)^T \mathbf{g}(\mathbf{x}) \right] \\ &\leq f_p(\mathbf{x}^*) - (\boldsymbol{\lambda}^*)^T \mathbf{h}(\mathbf{x}^*) - (\boldsymbol{\mu}^*)^T \mathbf{g}(\mathbf{x}^*) \\ &= f_p(\mathbf{x}^*) - (\boldsymbol{\mu}^*)^T \mathbf{g}(\mathbf{x}^*) \\ &\leq f_p(\mathbf{x}^*) = p^* \end{aligned} \tag{3.49}$$

  The last inequality is due to Eq. (3.42). But as $d^* = p^*$ due to the strong duality, the inequalities need to be equalities, i.e., $(\boldsymbol{\mu}^*)^T \mathbf{g}(\mathbf{x}^*) = 0$ and complementarity holds.
- Sufficient conditions:

  If $\mathbf{x}^*$ and $\{\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*\}$ satisfy the KKT conditions, then they are feasible for the respective primal and dual problems. Also, as $f_p(\mathbf{x})$ and all $g_j(\mathbf{x})$ are convex and $h_j\mathbf{x})$ are linear, Lagrangian in Eq. (3.38) as a sum of convex

functions is also convex (see Section A.8.3), and stationarity in Eq. (3.45) indicates $\mathbf{x}^*$ is a global minimum, therefore

$$\begin{aligned} d^* &= f_d(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = L(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = f_p(\mathbf{x}^*) - (\boldsymbol{\lambda}^*)^T \mathbf{h}(\mathbf{x}^*) - (\boldsymbol{\mu}^*)^T \mathbf{g}(\mathbf{x}^*) \\ &= f_p(\mathbf{x}^*) = p^* \end{aligned} \tag{3.50}$$

where the second line is due to the KKT conditions in Eqs. (3.46) and (3.48).

In summary, we see that the KKT conditions are necessary and satisfied by all solutions of constrained optimization problems, and they can also be sufficient for the solution of such problems if the primal problem further satisfies some additional conditions of convexity, such as in quadratic program if the quadratic objective function is also positive definite. In such a case, strong duality holds and the solution of the dual problem is the tightest lower bound of the primal, the same as the solution of the primal.

Consider, as an example of the inequality constrained minimization problems, the support vector machine for binary classification (Chapter 14). To find the decision boundary that optimally separates two classes, we need to solve the following minimization problem constrained by a set of inequalities imposed by the data samples $\{(\mathbf{x}_n, y_n),\ n = 1, \cdots, N\}$ in the training set:

$$\begin{cases} \text{min:} & \frac{1}{2}||\mathbf{w}||^2 \\ \text{s. t.:} & y_n(\mathbf{x}_n^T \mathbf{w} + b) \geq 1 \quad (n = 1, \cdots, N) \end{cases} \tag{3.51}$$

where $\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n$. We note that the quadratic objective function is convex and so are the affine inequality constraints, i.e., this is a convex problem. The solution $\mathbf{w}$ with minimum norm $||\mathbf{w}||^2$ subject to the inequality constraints can be found by solving the dual problem in terms of the Lagrange multipliers in $\boldsymbol{\alpha} = [\alpha_1, \cdots, \alpha_N]^T$, subject to certain inequality constraints (detailed derivation in Section 14):

$$\begin{cases} \text{max:} & \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \left(\mathbf{x}_n^T \mathbf{x}_m\right) \\ \text{s. t.:} & \sum_{n=1}^{N} \alpha_n y_n = \mathbf{y}^T \boldsymbol{\alpha} = 0, \quad \alpha_n \geq 0 \quad (n = 1, \cdots, N) \end{cases} \tag{3.52}$$

As the dual function can be written a quadratic form:

$$\boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{Q} \boldsymbol{\alpha} \tag{3.53}$$

where $\mathbf{Q}$ is an $N \times N$ symmetric matrix of which the component in the mth row and nth column is $Q(m, n) = y_m y_n \mathbf{x}_m^T \mathbf{x}_n$, this is a quadratic programming problem, which can be solved by the interior point algorithm to be discussed in Section 3.7.

3.4 Linear Programming (LP)

The basic problem in linear programming (LP) is to minimize/maximize a linear objective function $f(x_1, \cdots, x_N) = \sum_{i=1}^{N} c_i x_i$ of N variables $x_1, \cdots, x_N$ under a set of M linear constraints:

$$\begin{aligned} &\text{max:} && f(x_1, \cdots, x_N) = \textstyle\sum_{i=1}^{N} c_i x_i \\ &\text{s. t.:} && \begin{cases} \sum_{i=1}^{N} a_{1i} x_i \le b_1 \\ \cdots\cdots\cdots\cdots \\ \sum_{i=1}^{N} a_{Mi} x_i \le b_M \\ x_1 \ge 0, \cdots, x_N \ge 0 \end{cases} \end{aligned} \tag{3.54}$$

The LP problem can be more concisely represented in matrix form:

$$\begin{aligned} &\text{max:} && \mathbf{c}^T \mathbf{x} \\ &\text{s. t.:} && \begin{cases} \mathbf{A}\mathbf{x} - \mathbf{b} \le \mathbf{0} \\ \mathbf{x} \ge \mathbf{0} \end{cases} \end{aligned} \tag{3.55}$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_N \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_M \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \cdots & a_{MN} \end{bmatrix} \tag{3.56}$$

and the smaller or greater than sign in a vector inequality should be understood as an element-wise relationship between the corresponding elements of the vectors on both sides.

For example, the objective function could be the total profit, the constraints could be some limited resources. Specifically, $x_1, \cdots, x_N$ may represent the quantities of N different types of products, $c_1, \cdots, c_N$ represent their unit prices, and a_{ij} represent the consumption of the ith resource by the jth product, the goal is to maximize the profit $f(x_1, \cdots, x_N)$ subject to the constraints imposed by the limited resources $b_1, \cdots, b_M$.

Here all variables are assumed to be non-negative. If there exists a variable that is not restricted, it can be eliminated. Specifically, we can solve one of the constraining equations for the variable and use the resulting expression of the variable to replace all its appearances in the problem. For example:

$$\begin{aligned} &\text{max:} && f(x_1, x_2, x_3) = 2x_1 - x_2 + 3x_3 \\ &\text{s. t.:} && \begin{cases} x_1 - 2x_2 + x_3 = 3 \\ 3x_1 - x_2 + 4x_3 = 10 \\ x_2 \ge 0, \;\; x_3 \ge 0 \end{cases} \end{aligned} \tag{3.57}$$

Solving the first constraining equation for the free variable x_1, we get $x_1 = 3+2x_2-x_3$. Substituting this into the objective function and the other constraint, we get $2x_1 - x_2 + x_3 = 3x_2 + x_3 + 6$ and $5x_2 + x_3 = 1$, the problem can be

reformulated as:

$$\begin{array}{ll} \text{max:} & f(x_2, x_3) = 3x_2 + x_3 + 6 \\ \text{s. t.:} & \begin{cases} 5x_2 + x_3 = 1 \\ x_2 \geq 0, \;\; x_3 \geq 0 \end{cases} \end{array} \tag{3.58}$$

Given the primal LP problem above, we can further find its dual problem. We first construct the Lagrangian of primal problem:

$$L(\mathbf{x}) = \mathbf{c}^T\mathbf{x} - \mathbf{y}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = (\mathbf{c} - \mathbf{A}^T\mathbf{y})^T\mathbf{x} + \mathbf{y}^T\mathbf{b} \tag{3.59}$$

where $\mathbf{y} = [y_1, \cdots, y_M]^T$ contains the Lagrange multipliers for the inequality constraints $\mathbf{A}\mathbf{x} - \mathbf{b} \leq \mathbf{0}$, and $\mathbf{y} \geq \mathbf{0}$ according to Table 3.21. We define the dual function as the maximum of $L(\mathbf{x})$ over $\mathbf{x}$:

$$f_d(\mathbf{y}) = \max_{\mathbf{x}} L(\mathbf{x}) = \max_{\mathbf{x}}[\mathbf{c}^T\mathbf{x} - \mathbf{y}^T(\mathbf{A}\mathbf{x} - \mathbf{b})] = \max_{\mathbf{x}}[(\mathbf{c} - \mathbf{A}^T\mathbf{y})^T\mathbf{x} + \mathbf{y}^T\mathbf{b}] \tag{3.60}$$

To find the value of $\mathbf{x}$ that maximizes $L(\mathbf{x})$, we set its gradient to zero and get:

$$\bigtriangledown_{\mathbf{x}} L(\mathbf{x}) = \bigtriangledown_{\mathbf{x}}[(\mathbf{c} - \mathbf{A}^T\mathbf{y})^T\mathbf{x} + \mathbf{y}^T\mathbf{b}] = \mathbf{c} - \mathbf{A}^T\mathbf{y} = \mathbf{0} \tag{3.61}$$

Substituting this into $L_d(\mathbf{y}) = \max_{\mathbf{x}} L(\mathbf{x})$ we get: $f_d(\mathbf{y}) = \mathbf{y}^T\mathbf{b}$, which is the upper bound of $f_p(\mathbf{x})$, under the constraint that $\mathbf{c} - \mathbf{A}^T\mathbf{y} \leq 0$ so that $(\mathbf{c} - \mathbf{A}^T\mathbf{y})^T\mathbf{x} \leq 0$ contributes negatively to $L(\mathbf{x})$ (as $\mathbf{x} \geq 0$):

$$f_d(\mathbf{y}) = \mathbf{b}^T\mathbf{y} \geq L(\mathbf{x}) = \mathbf{c}^T\mathbf{x} - \mathbf{y}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) \geq \mathbf{c}^T\mathbf{x} = f_p(\mathbf{x}) \tag{3.62}$$

This upper bound needs to be minimized with respect to $\mathbf{y}$, to become the tightest upper bound, i.e., the original primal problem of maximization in Eq. (3.55) has now been converted into the following dual problem of minimization:

$$\begin{cases} \max: & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} & \mathbf{A}\mathbf{x} - \mathbf{b} \leq \mathbf{0}, \;\; \mathbf{x} \geq \mathbf{0} \end{cases} \iff \begin{cases} \min & \mathbf{b}^T\mathbf{y} \\ \text{s.t.} & \mathbf{A}^T\mathbf{y} - \mathbf{c} \geq \mathbf{0}, \;\; \mathbf{y} \geq \mathbf{0} \end{cases} \tag{3.63}$$

Following the same approach, we can also find the dual of a minimization primal problem:

$$\begin{aligned} \begin{cases} \min: & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} & \mathbf{A}\mathbf{x} - \mathbf{b} \leq \mathbf{0}, \;\; \mathbf{x} \geq \mathbf{0} \end{cases} &\iff \begin{cases} \max & \mathbf{b}^T\mathbf{y} \\ \text{s.t.} & \mathbf{A}^T\mathbf{y} - \mathbf{c} \leq \mathbf{0}, \;\; \mathbf{y} \leq \mathbf{0} \end{cases} \\ &\iff \begin{cases} \max & -\mathbf{b}^T\mathbf{y} \\ \text{s.t.} & \mathbf{A}^T\mathbf{y} + \mathbf{c} \geq \mathbf{0}, \;\; \mathbf{y} \geq \mathbf{0} \end{cases} \end{aligned} \tag{3.64}$$

Moreover, we can show that the dual of a dual problem is the original primal problem.

If either the primal or the dual is feasible and bounded, so is the other, and they form a strong duality, the solution d^* of the dual problem is the same as the solution p^* of the primal problem. Also, as the primal and dual problems are completely symmetric.

An inequality constrained LP problem can be converted to an equality constrained problem by introducing *slack variables*:

$$\sum_{i=1}^{N} a_i x_i \le b \quad \Longrightarrow \quad \sum_{i=1}^{N} a_i x_i + s = b, \quad (s \ge 0) \tag{3.65}$$

so that it can be reformulated to take the *standard form*:

$$\begin{array}{ll} \text{max:} & z = f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} = \sum_{j=1}^{N} c_j x_j \\ \text{s. t.:} & \left\{ \begin{array}{llllll} \sum_{i=1}^{N} a_{1i}x_i & +s_1 & & & & = b_1 \\ \sum_{i=1}^{N} a_{2i}x_i & & +s_2 & & & = b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^{N} a_{Mi}x_i & & & & +s_M & = b_M \\ x_1 \ge 0, & \cdots, & x_N \ge 0, & s_1 \ge 0, & \cdots & s_M \ge 0 \end{array} \right. \end{array} \tag{3.66}$$

We further redefine the following:

- $\mathbf{x}$ is an $N+M$ dimensional augmented variable vector that includes the M slack variables $\{s_1, \cdots, s_M\}$ as well as the N original variables $\{x_1, \cdots, x_N\}$:

$$\mathbf{x} = [x_1,\ x_2, \cdots, x_N,\ s_1,\ s_2, \cdots, s_M]^T \tag{3.67}$$

- $\mathbf{A}$ is redefined as an $M \times (N+M)$ augmented coefficient matrix that includes coefficients for both types of variables:

$$\mathbf{A} = \left[\begin{array}{cccc|cccc} a_{11} & a_{12} & \cdots & a_{1N} & 1 & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & a_{1N} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} & 0 & \cdots & 0 & 1 \end{array}\right] = [\ \mathbf{A}_{M\times N} \mid \mathbf{I}_{M\times M}\ ] \tag{3.68}$$

  where $\mathbf{A}_{M\times N}$ are the coefficients for the N original variables $\{x_1, \cdots, x_N\}$, and the identity matrix $\mathbf{I}_{M\times M} = [\mathbf{e}_1, \cdots, \mathbf{e}_M]$ is for the unity coefficients of the M slack variables $\{s_1, \cdots, s_M\}$, each of which appears in the M constraints only once.

Now the LP problem can be expressed in the standard form (original form on the left, with $\mathbf{x}$ and $\mathbf{A}$ redefined on the right):

$$\begin{array}{lll} \text{max:} & \mathbf{c}^T\mathbf{x} & \\ \text{s. t.:} & \left\{ \begin{array}{l} \mathbf{A}\mathbf{x} + \mathbf{s} = \mathbf{b} \\ \mathbf{x} \ge \mathbf{0},\ \mathbf{s} \ge \mathbf{0} \end{array} \right. \end{array} \quad \text{or} \quad \begin{array}{lll} \text{max:} & \mathbf{c}^T\mathbf{x} & \\ \text{s. t.:} & \left\{ \begin{array}{l} \mathbf{A}\mathbf{x} = \mathbf{b} \\ \mathbf{x} \ge \mathbf{0} \end{array} \right. \end{array} \tag{3.69}$$

An LP problem can be viewed geometrically. If normalize $\mathbf{c}$ so that $||\mathbf{c}|| = 1$, then the objective function $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$ becomes the projection of $\mathbf{x}$ onto $\mathbf{c}$. Each of the M equality constraints in $\mathbf{A}\mathbf{x} = \mathbf{b}$ represents a hyper-plane in the N-D vector space perpendicular to its normal direction $\mathbf{a}_j = [a_{i1}, \cdots, a_{IN}]^T$:

$$\sum_{j=1}^{N} a_{ij}x_j = \mathbf{a}_j^T\mathbf{x} = b_i, \qquad (i = 1, \cdots, M) \tag{3.70}$$

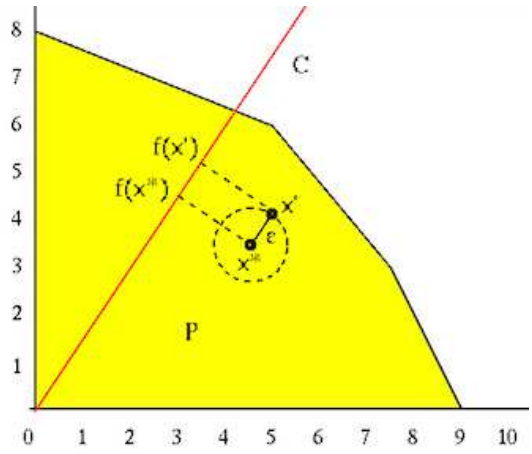


Figure 3.6 Fundamental Theorem of Linear Programming

Also, each of the N non-negativity conditions $x_j \geq 0$ in $\mathbf{x} \geq \mathbf{0}$ corresponds to a hyper-plane perpendicular to the jth standard basis vector $\mathbf{e}_j$. In general, in an N-D space, if none of the hyper-planes is parallel to any others, then no more than N hyper-planes intersect at one point. For example, in a 2-D or 3-D space, two straight lines or three surfaces intersect at a point. Here for the LP problem in the N-D space, the total number of intersection points formed by the $N + M$ hyper-planes is

$$C_{M+N}^{N} = C_{M+N}^{M} = \frac{(M+N)!}{N!\,M!} \tag{3.71}$$

The polytope enclosed by these $N + M$ hyper-planes is called the feasible region, in which the optimal solution must lie. The vertices of the polytopic feasible region are a subset of the C_{M+N}^{N} intersections.

Fundamental theorem of linear programming

The optimal solution $\mathbf{x}^*$ of a linear programming problem formulated above is either a vertex of the polytopic feasible region P, or lies on a hyper-surface of the polytope, on which all points are optimal solutions.

Proof:

Assume the optimal solution $\mathbf{x}^* \in P$ is interior to the polytopic feasible region P (as shown in Fig. 3.6), then there must exist some $\epsilon > 0$ such that the hyper-sphere of radius ϵ centered at $\mathbf{x}^*$ is inside P. Evaluate the objective function at a point on the hyper-sphere

$$\mathbf{x}' = \mathbf{x}^* + \epsilon\, \mathbf{c}/||\mathbf{c}|| \in P \tag{3.72}$$

we get

$$\begin{aligned} f(\mathbf{x}') &= \mathbf{c}^T\mathbf{x}' = \mathbf{c}^T(\mathbf{x}^* + \epsilon\, \mathbf{c}/||\mathbf{c}||) \\ &= \mathbf{c}^T\mathbf{x}^* + \epsilon\, \mathbf{c}^T\mathbf{c}/||\mathbf{c}|| = \mathbf{c}^T\mathbf{x}^* + \epsilon\, ||\mathbf{c}|| > \mathbf{c}^T\mathbf{x}^* = f(\mathbf{x}^*) \end{aligned} \tag{3.73}$$

i.e., $\mathbf{x}^*$ cannot be the optimal solution as assumed. This contradiction indicates that an optimal solution $\mathbf{x}^*$ must be on the surface of P, either at one of its vertices or on one of its surfaces. Q.E.D.

Based on this theorem, the optimization of a linear programming problem could be solved by exhaustively checking each of the C_{N+M}^N intersections formed by N of the $N+M$ hyper-planes to find (a) whether it is feasible (satisfying all $N+M$ constraints), and, if so, (b) whether the objective function $f(\mathbf{x})$ is maximized at the point. Specifically, we choose N of the $N+M$ equations for the the constraints and solve this N-equation and N-unknown linear system to get the intersection point of N corresponding hyper-planes. This brute-force method is most straight forward, but the computational complexity is high when N and M, and therefore C_{N+M}^N, become large.

Example 3.2

$$\begin{array}{ll} \text{max:} & f_p(x_1, x_2) = 2x_1 + 3x_2 \\ \text{s. t.:} & \left\{ \begin{array}{l} 2x_1 + x_2 \le 18 \\ 6x_1 + 5x_2 \le 60 \\ 2x_1 + 5x_2 \le 40 \\ x_1 \ge 0, \;\; x_2 \ge 0 \end{array} \right. \end{array} \quad \text{or} \quad \begin{array}{ll} \text{max:} & f_p(\mathbf{x}) = \mathbf{c}^T\mathbf{x} \\ \text{s. t.:} & \left\{ \begin{array}{l} \mathbf{Ax} - \mathbf{b} \le \mathbf{0} \\ \mathbf{x} \ge \mathbf{0} \end{array} \right. \end{array}$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 18 \\ 60 \\ 40 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 2 & 1 \\ 6 & 5 \\ 2 & 5 \end{bmatrix}$$

This problem has $N = 2$ variables with the same number of non-negativity constraints and $M = 3$ linear inequality constraints, as shown in Fig. 3.7. The $n+m = 5$ straight lines form $C_{N+M}^N = C_5^2 = 10$ intersections, out of which 5 are the vertices of the polygonal feasible region satisfying all constraints. The value of the objective function $f_p(x_1, x_2) = \mathbf{c}^T\mathbf{x} = c_1x_1 + c_2x_2$ is proportional to the projection of the 2-D variable vector $\mathbf{x} = [x_1, \; x_2]^T$ onto the coefficient vector $\mathbf{c} = [2, \; 3]^T$. The goal of this linear programming problem is to find a point $\mathbf{x}$ inside the polygonal feasible region with maximum $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$ (the projection of $\mathbf{x}$ onto vector $\mathbf{c}$ if $||\mathbf{c}|| = 1$). Geometrically, this can be understood as to push the hyper-plane $\mathbf{c}^T\mathbf{x} = z$, here a line in 2-D space, its normal direction $\mathbf{c}$, away from the origin along to eventually arrive at the vertex of the polytopic feasible region farthest away from the origin.

This LP problem can be converted into the standard form:

$$\begin{array}{ll} \text{max:} & f_p(x_1, x_2) = 2x_1 + 3x_2 \\ \text{s. t.:} & \left\{ \begin{array}{l} 2x_1 + x_2 \le 18 \\ 6x_1 + 5x_2 \le 60 \\ 2x_1 + 5x_2 \le 40 \\ x_1 \ge 0, \;\; x_2 \ge 0 \end{array} \right. \end{array} \quad \Longrightarrow \quad \begin{array}{ll} \text{max:} & f_p(x_1, x_2) = 2x_1 + 3x_2 \\ \text{s. t.:} & \left\{ \begin{array}{l} 2x_1 + x_2 + s_1 = 18 \\ 6x_1 + 5x_2 + s_2 = 60 \\ 2x_1 + 5x_2 + s_3 = 40 \\ x_1 \ge 0, \; x_2 \ge 0, \; s_1 \ge 0, \; s_2 \ge 0, \; s_3 \ge 0 \end{array} \right. \end{array}$$

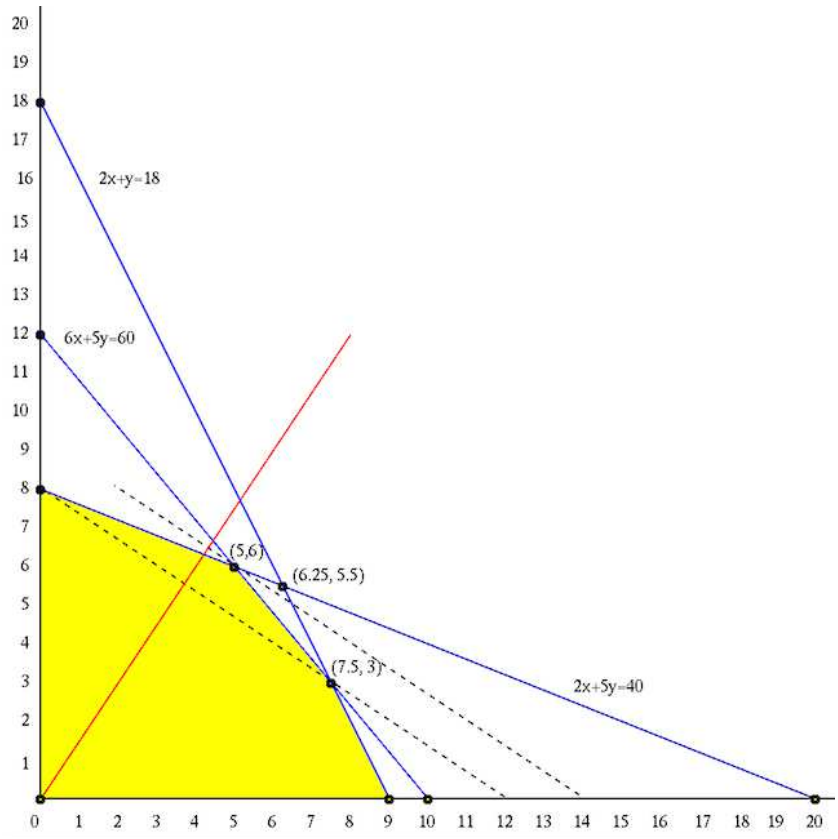


Figure 3.7 Simplex Algorithm Example (Original Problem)

Listed in the table below are the $C_{M+N}^{N} = C_5^2 = 10$ intersections $\mathbf{x}$, called basic solutions, together with the corresponding slack variables s_1, s_2, s_3 and the objective function value $f(\mathbf{x})$ (proportional to the projection of $\mathbf{x}$ onto $\mathbf{c}$). We see that out of the ten basic solutions, five are feasible (vertices of the polytopic feasible region) with all $M + N = 5$ variables taking non-negative values, while the remaining five are not feasible, as some of the slack variables are negative, and thereby violating the constraints. Out of the five feasible solutions, the one at $\mathbf{x}^* = (5,\ 6)$ is optimal with maximum objective function value $f_p(x_1, x_2) = 2x_1 + 3x_2 = 2 \times 5 + 3 \times 6 = 28$.

	x_1	x_2	s_1	s_2	s_3	feasibility	$\mathbf{c}^T\mathbf{x}/\|\|\mathbf{c}\|\| = 1$
1	5	6	2	0	0	feasible	28/7.77
2	6.25	5.5	0	−5	0	infeasible	29/8.04
3	7.5	3	0	0	10	feasible	24/6.66
4	20	0	−2	−60	20	infeasible	40/11.1
5	10	0	−2	0	20	infeasible	20/5.55
6	9	0	0	6	22	feasible	18/4.10
7	0	8	10	20	0	feasible	24/6.66
8	0	12	6	0	−20	infeasible	36/9.98
9	0	18	0	−30	−50	infeasible	54/14.99
10	0	0	18	60	40	feasible	0/0.00

This primal maximization problem can be converted into the dual minimization problem:

$$\text{min:} \quad f_d(y_1, y_2, y_3) = 18y_1 + 60y_2 + 40y_3$$
$$\text{s. t.:} \quad \begin{cases} 2y_1 + 6y_2 + 2y_3 \geq 2 \\ y_1 + 5y_2 + 5y_3 \geq 3 \\ y_1 \geq 0,\ y_2 \geq 0,\ y_3 \geq 0 \end{cases}$$

or

$$\text{min:} \quad f_d(\mathbf{y}) = \mathbf{b}^T\mathbf{y}$$
$$\text{s. t.:} \quad \begin{cases} \mathbf{A}^T\mathbf{y} - \mathbf{c} \geq \mathbf{0} \\ \mathbf{y} \geq \mathbf{0} \end{cases}$$

As shown in Fig. 3.8, the five planes, including $m = 2$ from the constraining inequalities and $n = 3$ for the non-negative constraints $y_1 \leq 0$, $y_2 \leq 0$, and $y_3 \leq 0$ for the three variables, form 10 intersection points:

	(y_1, y_2, y_3)	feasibility	objective function
1	$(1, 0, 0)$	infeasible	18
2	$(3, 0, 0)$	feasible	54
3	$(0, 1/3, 0)$	infeasible	20
4	$(0, 3/5, 0)$	feasible	36
5	$(0, 0, 1)$	feasible	40
6	$(0, 0, 3/5)$	infeasible	24
7	$(0, 1/5, 2/5)$	feasible	28
8	$(1/2, 0, 1/2)$	feasible	29
9	$(-2, 1, 0)$	infeasible	24
10	$(0, 0, 0)$	infeasible	0

We see that at the feasible point $(0, 1/5, 2/5)$, the dual function $f_d(y_1, y_2, y_3) = 18y_1 + 60y_2 + 40y_3$ is minimized to be 28, the same as the maximum of the primal function $f_p(x_1, x_2) = 2x_1 + 3x_2$ at $(5, 6)$.

3.5 The Simplex Algorithm

The simplex algorithm finds the optimal solution of a LP problem by an iterative process that traverses along a sequence of edges of the polytopic feasible region,

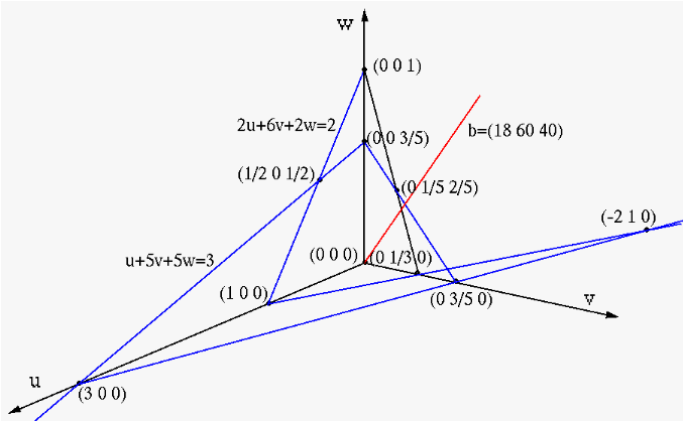


Figure 3.8 Simplex Algorithm Example (Dual Problem)

starting at the origin and through a sequence of vertices $\mathbf{x}$ with progressively greater objective value $f(\mathbf{x})$, until eventually reaching the optimal solution. By doing so, it avoids checking exhaustively all vertices of the feasible region for optimality.

We first consider the equality constraint $\mathbf{Ax} = [\mathbf{A}_{M\times N} \mid \mathbf{I}_{M\times M}]\mathbf{x} = \mathbf{b}$ of the standard LP problem. This is an under-determined linear system of $N + M$ variables but only M equations, only M of its $N + M$ columns of the coefficient matrix $\mathbf{A}$ are independent (assuming rank$(\mathbf{A}) = M$). Initially we choose the M columns of the identity coefficient matrix $\mathbf{I}$ as the independent columns, but subsequently we can choose any other M columns of $\mathbf{A}$ as the independent columns and convert them into a standard basis vector by Gauss-Jordan elimination together with the corresponding variables in $\mathbf{x}$. The resulting equation $\mathbf{Ax} = \mathbf{b}$ remains equivalent to the original one, i.e., the M equality constraints are always preserved and never violated in the process.

For convenience of discussion and without loss of generality, we could reorder (not actually carried out) the $N + M$ columns in $\mathbf{A}$, together with the corresponding variables in $\mathbf{x}$, so that the constraining equation would always takes the following form:

$$\mathbf{Ax} = [\mathbf{A}_n \;\; \mathbf{I}] \begin{bmatrix} \mathbf{x}_n \\ \mathbf{x}_b \end{bmatrix} = \mathbf{A}_n\mathbf{x}_n + \mathbf{I}\,\mathbf{x}_b = \mathbf{b} \tag{3.74}$$

Now the variables in the M-D vector $\mathbf{x}_b$ corresponding to the M independent columns in $\mathbf{I}$ are the *basic variables*, while the N-D vector $\mathbf{x}_n$ corresponding to the remaining N dependent columns, now denoted by $\mathbf{A}_n$, are the *non-basic variables*. This equation will always hold if $\mathbf{x}_n = \mathbf{0}$ and $\mathbf{x}_b = \mathbf{b}$. Such a solution is called a *basic solution* of the linear system $\mathbf{Ax} = \mathbf{b}$:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_n \\ \mathbf{x}_b \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b} \end{bmatrix} \tag{3.75}$$

Initially, $\mathbf{A}_n$ is the coefficient matrix in the inequality constraint $\mathbf{Ax} \leq \mathbf{b}$, and the corresponding non-basic variables $\mathbf{x}_n = \mathbf{0}$ is actually the origin. But through the iteration, we will keep converting some other columns of $\mathbf{A}$ into standard basis vectors $\{\mathbf{e}_1, \cdots, \mathbf{e}_M\}$, the column vectors of $\mathbf{I}$, by Gauss-Jordan elimination. The variables in $\mathbf{x}$ corresponding to the new standard basis vectors become the basic variables, while those corresponding to columns that become non-standard basis vectors become non-basis variables.

The basic solutions corresponding the C_{N+M}^N ways to choose M of the $N+M$ columns of $\mathbf{A}$ as independent are actually the intersections formed by any N of the $N+M$ constraining hyper-planes in the N-D space. As noted before, only a subset of these C_{N+M}^N basic solutions satisfy the M constraints, and they are called *basic feasible solutions (BFS)*. The goal of the iterative process of the simplex algorithm is to find the optimal basic feasible solution that maximizes $f(\mathbf{x})$ without exhausting all C_{N+M}^N possibilities. This is done by selecting the columns in such a way that the value of the objective function will always be maximally increased, until eventually we find the optimal solution $\mathbf{x}^*$ at one of the vertex of the polytopic feasible region.

Specially, the implementation of the simplex algorithm is based on a tableau with $N+M+1$ columns and $M+1$ rows, initialized as below:

- The first M rows are for the coefficients in the equality constraints $\sum_{j=1}^N a_{ij}x_j + s_i = b_i$ $(i = 1, \cdots, M)$. An additional $(M+1)st$ row at the bottom is for the coefficients in the objective function $f(\mathbf{x}) = \sum_{j=1}^N c_j x_j$, which is zero initially at the origin $x_1 = \cdots = x_N = 0$, i.e., $z - c_1 x_1 - \cdots - c_N x_N = 0$.
- The first $N+M$ columns contain the coefficients for the N original variables $\{x_1, \cdots, x_N\}$ and the M slack variables $\{s_1, \cdots, s_M\}$, and an additional $(N+M+1)st$ column is for the constants $\mathbf{b} = [b_1, \cdots, b_M]^T$ on the right-hand side of the M constraint equations. The last element of the column is the right-hand side of the objective equation $z = f(\mathbf{x})$, which is initially zero.

basic variables	x_1	x_2	$\cdots$	x_N	s_1	s_2	$\cdots$	s_M	basic solution
s_1	a_{11}	a_{12}	$\cdots$	a_{1N}	1	0	$\cdots$	0	b_1
s_2	a_{21}	a_{22}	$\cdots$	a_{2N}	0	1	$\cdots$	0	b_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
s_M	a_{M1}	a_{M2}	$\cdots$	a_{MN}	0	0	$\cdots$	1	b_M
z	$-c_1$	$-c_2$	$\cdots$	$-c_M$	0	0	$\cdots$	0	0

(3.76)

At this initial stage of the iteration, the N original variables $\mathbf{x}_n = [x_1, \cdots, x_N]^T$ are the non-basic variables, while the M slack variables $\mathbf{x}_b = [s_1, \cdots, s_M]^T$ are the basic variables and their coefficients form an $M \times M$ identity matrix $\mathbf{I}$, composed of the standard basis vectors. The corresponding feasible basic solution is

simply

$$\mathbf{x}_b = [s_1, \cdots, s_M]^T = \mathbf{b}, \quad \mathbf{x}_n = [x_1, \cdots, x_N]^T = \mathbf{0} \tag{3.77}$$

that satisfies all the constraints $\mathbf{Ax} = \mathbf{A}_n\mathbf{x}_n + \mathbf{I}\mathbf{x}_b = \mathbf{b}$.

In each of the subsequent iterations, we will select one of the non-basic variables, called the *entering variable*, to replace one of the basic variables, called the *leaving variable*, in such a way that the value of the objective function value $z = f(\mathbf{x})$ in the last row will be maximally increased, while all constraints remain satisfied. Here are the steps in each iteration:

- **Selection of the entering variable**

  This section is based on the maximization of $z = f(\mathbf{x})$. We select x_j in the jth column of the tableau if $-c_j$ is most negative, i.e., x_j is most heavily weighted by $c_j = \max\{c_1, \cdots, c_N\}$, so that it will increase $z = \sum_{j=1}^{N} c_j x_j$ more than any other x_k $(k \neq j)$.
- **Selection of the leaving variable**

  This selection is based on the constraints imposed on the selected entering variable x_j. In general, the restriction on x_j set by the kth constraint $\sum_{j=1}^{N} a_{kj}x_j \leq b_k$ is $x_j \leq b_k/a_{kj}$ when $a_{kj} > 0$. If $b_i/a_{ij} = \min\{b_k/a_{1j}, \cdots, b_k/a_{Mj}\}$, then the ith constraint is most restrictive on x_j, we will therefore select the corresponding basic variable s_i as the leaving variable, i.e., it becomes a non-basic variable to be set to zero, so that x_j can be maximally increased without violating the constraints. If all $a_{kj} < 0$ for all $k = 1, \cdots, M$, variable x_j is not bounded.
- **Gauss-Jordan elimination based on Pivoting**

  To convert the entering variable x_j to a basic variable to replace the leaving variable s_i, we need to turn the corresponding jth column into a standard basis vector $\mathbf{e}_i$. This is realized by pivoting on a_{ij} in the following steps:

  1.Divide the pivot row $\mathbf{r}_i$ by a_{ij}: $\mathbf{r}_i \leftarrow \mathbf{r}_i/a_{ij}$. Now $a_{ij} = 1$.

  2.Subtract $a_{kj}\mathbf{r}_i$ from the kth row: $\mathbf{r}_k \leftarrow \mathbf{r}_k - a_{kj}\mathbf{r}_i$ so that $a_{kj} = 0$ for all $k = 1, \cdots, M+1$, $k \neq i$, including the last row $\mathbf{r}_{M+1}$.

  Now the jth column corresponding to the entering variable x_j becomes a standard basis vector $\mathbf{e}_i$, i.e., x_j becomes a basic variable that takes the value of b_i, and the column corresponding to the leaving variable s_i is no longer a standard basis vector, and $s_i = 0$ as it is now a non-basic variable.

As Gauss-Jordan elimination converts the linear equations to a set of equivalent equations, the constraints remain satisfied through out the process. Although the membership of the basic and non-basic variable groups keeps changing in the iteration, so long as $\mathbf{x}_b = \mathbf{b}$ and $\mathbf{x}_n = \mathbf{0}$, the M constraint equations $\mathbf{Ax} = \mathbf{A}_n\mathbf{x}_n + \mathbf{I}\mathbf{x}_b = \mathbf{b}$ always holds.

This iterative process keeps replacing the slack variables $\{s_1, \cdots, s_M\}$ (the basic variables initially) by the original variables $\{x_1, \cdots, x_N\}$ (the non-basic

variables initially), one at a time, until all original variables become basic variables and all elements in the last row are non-negative.

The final result can be read out directly from the tableau. The variables corresponding to the M standard basis vectors $\mathbf{e}_i$ $(i = 1, \cdots, M)$ are the final basic variables that take the values in $\mathbf{b}$ in the right-most column of the tableau. They form the optimal basic solution, with the maximum of the objective function $z = f(\mathbf{x})$ given by the last element also in the last column. The remaining n variables corresponding to non-standard columns are non-basic variables that take the value zero. When $M > N$ (more constraints than variables), some of the slack variables may remain in the basic variable group taking non-zero values; when $M < N$, some of the original variable may be in the non-basic group taking the value zero.

Example 3.3 We re-solve the LP problem considered in the previous examples, now in the standard form:

$$\begin{aligned} \text{max:} \quad & f(\mathbf{x}) = 2x_1 + 3x_2 \\ \text{s. t.:} \quad & \begin{cases} 2x_1 + x_2 + s_1 = 18 \\ 6x_1 + 5x_2 + s_2 = 60 \\ 2x_1 + 5x_2 + s_3 = 40 \\ x_1 \geq 0,\ x_2 \geq 0,\ s_1 \geq 0,\ s_2 \geq 0,\ s_3 \geq 0 \end{cases} \end{aligned}$$

The standard form is further converted to a tableau as shown below. The left-most column indicates the basic variables, the next $N = 2$ columns are for the variables x and y,the next $M = 3$ columns are for the slack variables u, v, and w, the right most column is for the constants b_1, b_2, and b_3 on the right-hand side of the equations.

	x_1	x_2	s_1	s_2	s_3	b
s_1	2	1	1	0	0	18
s_2	6	5	0	1	0	60
s_3	2	5	0	0	1	40
z	-2	-3	0	0	0	0

In this initial state, the basic variables are $s_1 = 18$, $s_2 = 60$, and $s_3 = 40$, the non-basic variables are $x_1 = x_2 = 0$. The corresponding basic feasible solution is at the origin.

- Select column $j = 2$ as the pivot column, as $-c_2 = -3 < -c_1$ is the most negative coefficient.
- Select $\mathbf{r}_3$ as the pivot row, as $b_3/a_{32} = 40/5 = 8 = \min\{18/1 = 18,\ 60/5 = 12,\ 40/5 = 8\}$.
- Divide pivot row $\mathbf{r}_i = \mathbf{r}_3 = [2,\ 5,\ 0,\ 0,\ 1,\ 40]$ by the pivot element $a_{ij} = 32 = 5$ to get $\mathbf{r}_i = \mathbf{r}_3 = [0.4,\ 1,\ 0,\ 0,\ 0.2,\ 8]$.
- Subtract $a_{kj}\mathbf{r}_k = a_{k2}\mathbf{r}_k$ from row $\mathbf{r}_k$, so that $a_{kj} = a_{k2} = 0$, $(k = 1, \cdots, M + 1 = 4,\ k \neq i)$, including the last row $\mathbf{r}_4$.

	x_1	x_2	s_1	s_2	s_3	b
s_1	1.6	0	1	0	-0.2	10
s_2	4	0	0	1	-1	20
x_2	0.4	1.0	0	0	0.2	8
z	-0.8	0	0	0	0.6	24

The entering variable $x_j = x_2$ becomes a basic variable, and the leaving variable $s_i = s_3$ becomes a non-basic variable, replaced by new basic variable x_2. The corresponding basic feasible solution is at $x_1 = 0,\ x_2 = b_3 = 8$, with $s_1 = 10$, $s_2 = 20$, $s_3 = 0$. The objective function value is $z = \mathbf{c}^T\mathbf{x} = 24$.

- Select column $j = 1$ as the pivot column, as $-c_1 = -0.8$ is the most (only) negative coefficient in the last row.
- Select row $i = 2$ as the pivot row, as the ratio $b_2/a_{21} = 20/4 = 5 = \min\{10/1.6 = 6.25,\ 20/4 = 5,\ 8/0.4 = 20\}$.
- Divide pivot row $\mathbf{r}_i = \mathbf{r}_2 = [4,\ 0,\ 0,\ 1,\ -1,\ 20]$ by the pivot element $a_{ij} = a_{21} = 4$ to get $\mathbf{r}_i = \mathbf{r}_2 = [1,\ 0,\ 0,\ 0.25,\ -0.25,\ 5]$.
- Subtract $a_{kj}\mathbf{r}_k = a_{k1}\mathbf{r}_k$ from row $\mathbf{r}_k$, so that $a_{kj} = a_{k1} = 0, (k = 1, \cdots, M + 1 = 4)$, including the last row $\mathbf{r}_4$.

	x_1	x_2	s_1	s_2	s_3	b
s_1	0	0	1	-0.4	0.2	2
x_1	1	0	0	0.25	-0.25	5
x_2	0	1	0	$-0,1$	0.3	6
z	0	0	0	0.2	0.4	28

The entering variable x_1 becomes a basic variable, and the leaving variable s_1 becomes a non-basic variable, replaced by the new basic variable x_1. The corresponding basic feasible solution is at $x_1 = b_2 = 5,\ x_2 = b_3 = 6$, with $s_1 = 2,\ s_2 = s_3 = 0$. The objective function value is $z = \mathbf{c}^T\mathbf{x} = 28$. Now that both x_1 and x_2 have become basic variables, the optimal solution has been obtained.

This problem has $C_{N+M}^N = C_5^2 = 10$ basic solutions, corresponding to the same number of intersections, out of which five are feasible, as previously obtained. The simplex method finds three of these feasible solutions, starting from $x_1 = x_2 = 0$ at the origin with $z = 0$, through the vertex at $x_1 = 0$ and $x_2 = 8$ with $z = 24$, to the optimal solution at $x_1 = 5$ and $x_2 = 6$ with $z = 28$.

The Matlab code for implementing the simplex algorithm is listed below. The function takes input including matrix $\mathbf{A}$ as well as the vectors $\mathbf{b}$ and $\mathbf{c}$, and generates the optimal solution $\mathbf{x}^*$, the corresponding maximum value of the objective function $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}^*$, together with the values of the slack variable $\mathbf{s}$.

```
function [x s z] = MySimplex(A,b,c)
    [m n]=size(A)          % coefficient matrix for variables x
    x=zeros(n,1);          % initial zero values for x
    s=zeros(m,1);
    A=[A eye(m) b; -c zeros(1,m+1)]; % initialization of tableau

    [cmin, pc]=min( A(m+1,1:n+m) );  % pc, index of first pivot column
    it=0;
    while cmin<0           % iteration until all c's in last row are zero
        it=it+1;
        rmin=9e9;
        for i=1:m          % find pivot row
            if A(i,n+m+1)~=0 & A(i,pc)>0
                w=A(i,n+m+1)./A(i,pc);
                if w<rmin
                    pr=i;  % pr, index of the pivot row
                    rmin=w;
                end
            end
        end
        p=A(pr,pc);        % get the pivot element
        A(pr,:)=A(pr,:)/p; % modify pivot row
        for i=1:m+1        % modify all k+1 rows
            if i~=pr
                A(i,:)=A(i,:)-A(pr,:)*A(i,pc);
            end
        end
        z=A(m+1,n+m+1);    % maximum value of objective function so far

        [cmin, pc]=min( A(m+1,1:n+m) ); % pivot column of next iteration
    end
end
```

3.6 Quadratic Programming (QP)

A quadratic programming (QP) problem is to minimize a quadratic function subject to some equality and/or inequality constraints:

$$\begin{aligned} \text{min:} \quad & f(\mathbf{x}) = \tfrac{1}{2}[\mathbf{x}-\mathbf{m}]^T\mathbf{Q}[\mathbf{x}-\mathbf{m}] = \tfrac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} + c \\ \text{s. t.:} \quad & \mathbf{A}\mathbf{x} \le \mathbf{b} \end{aligned} \tag{3.78}$$

where $\mathbf{Q} = \mathbf{Q}^T$ is an $N \times N$ positive definite matrix, $\mathbf{A}$ an $M \times N$ matrix, and $\mathbf{b}$ is an M-D vector. Also, $\mathbf{c} = -\mathbf{Q}\mathbf{m}$, $c = \mathbf{m}^T\mathbf{Q}\mathbf{m}/2$. Here the scalar constant c

can be dropped as it does not play any role in the optimization. Note that $\mathbf{x}^T\mathbf{Q}\mathbf{x}$ is a hyper elliptic paraboloid with the minimum at point $\mathbf{m}$ in the N-D space.

We first consider the special case where the QP problem is only subject to equality constraints and we assume $m \leq N$, i.e., the number of constraints is smaller than the number of unknowns in $\mathbf{x}$. Then the solution $\mathbf{x}$ has to satisfy $\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$, i.e., it has to be on M hyper planes in the N-D space.

The Lagrangian function of the QP problem is:

$$L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \boldsymbol{\lambda}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} + \boldsymbol{\lambda}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) \tag{3.79}$$

To find the optimal solution, we first equate the derivatives of the Lagrangian with respect to both $\mathbf{x}$ and $\boldsymbol{\lambda}$ to zero:

$$\begin{aligned} \bigtriangledown_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}) &= \mathbf{Q}\mathbf{x} + \mathbf{c} + \mathbf{A}^T\boldsymbol{\lambda} = \mathbf{0} \\ \bigtriangledown_{\boldsymbol{\lambda}} L(\mathbf{x}, \boldsymbol{\lambda}) &= \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0} \end{aligned} \tag{3.80}$$

These two equations can be combined and expressed in matrix form as:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} -\mathbf{c} \\ \mathbf{b} \end{bmatrix} \tag{3.81}$$

We then solve this system of $m + N$ equations to get the optimal solution $\mathbf{x}^*$ and the corresponding $\boldsymbol{\lambda}^*$.

Example 3.4

$$\begin{aligned} \text{min:} \quad & f(x_1, x_2) = x_1^2 + x_2^2 = \tfrac{1}{2}[x_1,\ x_2] \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \tfrac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} \\ \text{s. t.:} \quad & \mathbf{A}\mathbf{x} = [1,\ 1] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 + x_2 = \mathbf{b} = 1 \end{aligned}$$

where

$$\mathbf{Q} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \quad \mathbf{A} = [1,\ 1], \quad \mathbf{c} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{b} = 1;$$

$$\begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Solving this equation we get the solution $x_1^* = x_2^* = 0.5$ and $\lambda^* = -1$, at which the function $f(x_1^*, x_2^*) = 0.5$ is minimized subject to $x_1 + x_2 = 1$.

If $M = N$, i.e., the number of equality constraints is the same as the number of variables, then the variable $\mathbf{x}$ is uniquely determined by the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, as the intersect of $M = N$ hyper planes, independent of the objective function $f(\mathbf{x})$. Further if $M > N$, i.e., the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is over constrained,

and its solution does not exist in general. It is therefore more interesting to consider QP problems subject to both equality and inequality constraints:

$$\begin{aligned} &\text{min:} && f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ &\text{s. t.:} && \mathbf{A}\mathbf{x} \leq \mathbf{b} \end{aligned}$$

3.7 Interior Point Methods

Consider the generic constrained optimization problem in the following form:

$$\begin{aligned} &\text{min:} && f(\mathbf{x}) \\ &\text{s. t.:} && \begin{cases} \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \quad \text{or} \quad \mathbf{g}(\mathbf{x}) \geq \mathbf{0} \\ \mathbf{x} \geq \mathbf{0} \end{cases} \end{aligned} \tag{3.82}$$

By introducing slack variables $\mathbf{s} \geq \mathbf{0}$, all inequality constraints can be converted into equality constraints:

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{0} \implies \mathbf{g}(\mathbf{x}) + \mathbf{s} = \mathbf{0}, \qquad \mathbf{g}(\mathbf{x}) \geq \mathbf{0} \implies \mathbf{g}(\mathbf{x}) - \mathbf{s} = \mathbf{0} \tag{3.83}$$

which can then be combined with the equality constraints and still denoted by $\mathbf{h}(\mathbf{x}) = \mathbf{0}$, while the slack variables $\mathbf{s}$ are also combined with the original variables and still denoted by $\mathbf{x}$. We assume there are in total M equality constraints and N variables. Now the optimization problem can be reformulated as:

$$\begin{aligned} &\text{min:} && f(\mathbf{x}) \\ &\text{s. t.:} && \begin{cases} \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ \mathbf{x} \geq \mathbf{0} \end{cases} \end{aligned} \tag{3.84}$$

The Lagrangian is

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) + \boldsymbol{\lambda}^T\mathbf{h}(\mathbf{x}) - \boldsymbol{\mu}^T\mathbf{x} \tag{3.85}$$

The KKT conditions are

$$\begin{cases} \bigtriangledown_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{g}_f(\mathbf{x}) + \mathbf{J}_{\mathbf{h}}^T(\mathbf{x})\boldsymbol{\lambda} - \boldsymbol{\mu} = \mathbf{0} & \text{(stationarity)} \\ \mathbf{h}(\mathbf{x}) = \mathbf{0}, \quad \mathbf{x} \geq \mathbf{0} & \text{(primal feasibility)} \\ \boldsymbol{\mu} \geq \mathbf{0} & \text{(dual feasibility)} \\ \mu_j x_j = 0, \quad (j = 1, \cdots, N) \quad \text{or} \quad \mathbf{XM1} = \mathbf{0} & \text{(complementarity)} \end{cases} \tag{3.86}$$

where

$$\mathbf{X} = \begin{bmatrix} x_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & x_n \end{bmatrix}_{N \times N}, \qquad \mathbf{M} = \begin{bmatrix} \mu_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mu_N \end{bmatrix}_{N \times N} \tag{3.87}$$

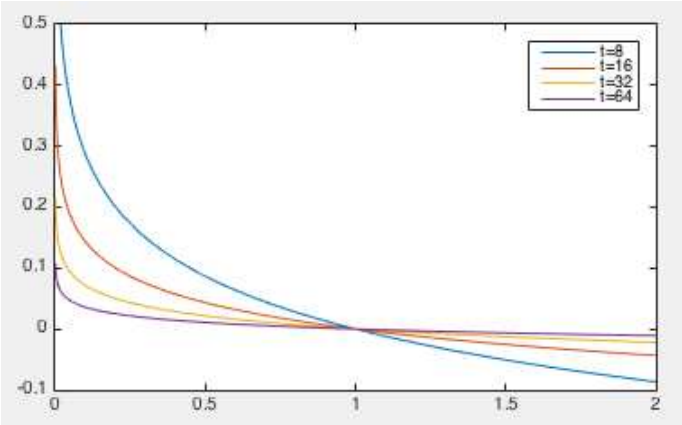


Figure 3.9 Logarithmic Barrier Function

and $\mathbf{J_f}(\mathbf{x})$ is the Jacobian matrix of $\mathbf{h}(\mathbf{x})$:

$$\mathbf{J_h}(\mathbf{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_m}{\partial x_1} & \cdots & \frac{\partial h_m}{\partial x_n} \end{bmatrix}_{M \times N} \tag{3.88}$$

To simplify the problem, the non-negativity constraints can be removed by introducing an indicator function:

$$I(x) = \begin{cases} 0 & x \geq 0 \\ \infty & x < 0 \end{cases} \tag{3.89}$$

which, when used as a cost function, penalizes any violation of the non-negativity constraint $x \geq 0$. Now the optimization problem can be reformulated as shown below without the non-negative constants $\mathbf{x} \geq \mathbf{0}$:

$$\begin{array}{ll} \text{min:} & f(\mathbf{x}) + \sum_{i=1}^{N} I(x_i) \\ \text{s. t.:} & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{array} \tag{3.90}$$

However, as the indicator function is not smooth and therefore not differentiable, it is approximated by the *logarithmic barrier function* which approaches $I(x)$ when the parameter $t > 0$ approaches infinity:

$$-\frac{1}{t}\ln(x) \quad \stackrel{t \to \infty}{\Longrightarrow} \quad I(x) \tag{3.91}$$

Now the optimization problem can be written as:

$$\begin{array}{ll} \text{min:} & f(\mathbf{x}) - \frac{1}{t}\sum_{i=1}^{N} \ln x_i \\ \text{s. t.:} & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{array} \tag{3.92}$$

The Lagrangian is:

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = f(\mathbf{x}) + \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{x}) - \frac{1}{t} \sum_{j=1}^{N} \ln x_j \tag{3.93}$$

and its gradient is:

$$\bigtriangledown_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{g}_f(\mathbf{x}) + \mathbf{J}_{\mathbf{h}}^T(\mathbf{x})\boldsymbol{\lambda} - \frac{1}{t}\mathbf{X}^{-1}\mathbf{1} \tag{3.94}$$

We define

$$\boldsymbol{\mu} = \frac{\mathbf{x}}{t} = \frac{1}{t}\mathbf{X}^{-1}\mathbf{1}, \quad \text{i.e.} \quad \mathbf{X}\mu = \mathbf{XM1} = \frac{\mathbf{1}}{t} \tag{3.95}$$

and combine this equation with the KKT conditions of the new optimization problem to get

$$\begin{cases} \mathbf{g}_f(\mathbf{x}) + \mathbf{J}_{\mathbf{h}}^T(\mathbf{x})\boldsymbol{\lambda} - \mu = \mathbf{0} & \text{(stationarity)} \\ \mathbf{h}(\mathbf{x}) = \mathbf{0} & \text{(primal feasibility)} \\ \mathbf{XM1} - \mathbf{1}/t = \mathbf{0} & \text{(complementarity)} \end{cases} \tag{3.96}$$

The third condition is actually from the definition of $\boldsymbol{\mu}$ given above, but here it is labeled as complementarity, so that these conditions can be compared with the KKT conditions of the original problem. We see that the two sets of KKT conditions are similar to each other, but with the following differences:

- The non-negativity $\mathbf{x} \geq \mathbf{0}$ in the primal feasibility of the original KKT conditions is dropped;
- Consequently the inequality $\boldsymbol{\mu} \geq \mathbf{0}$ in the dual feasibility of the original KKT conditions is also dropped;
- There is an extra term $\mathbf{1}/t$ in complementarity which will vanish when $t \to \infty$.

The optimal solution that minimizes the objective function $f(\mathbf{x})$ can be found by solving the simultaneous equations in the modified KKT conditions (with no inequalities) given above. To do so, we first express the equations in the KKT conditions above as a nonlinear equation system $\mathbf{F}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}$, and find its Jacobian matrix $\mathbf{J}_{\mathbf{F}}$ composed of the partial derivatives of the function $\mathbf{F}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$ with respect to each of the three variables $\mathbf{x}$, $\boldsymbol{\lambda}$, and $\boldsymbol{\mu}$:

$$\mathbf{F}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \begin{bmatrix} \mathbf{g}_f(\mathbf{x}) + \mathbf{J}_{\mathbf{h}}^T(\mathbf{x})\boldsymbol{\lambda} - \mu \\ \mathbf{h}(\mathbf{x}) \\ \mathbf{XM1} - \mathbf{1}/t \end{bmatrix}, \quad \mathbf{J}_{\mathbf{F}} = \begin{bmatrix} \mathbf{W}(\mathbf{x}) & \mathbf{J}_{\mathbf{h}}^T(\mathbf{x}) & -\mathbf{I} \\ \mathbf{J}_{\mathbf{h}}(\mathbf{x}) & \mathbf{0} & \mathbf{0} \\ \mathbf{M} & \mathbf{0} & \mathbf{X} \end{bmatrix} \tag{3.97}$$

where

$$\begin{aligned} \mathbf{W}(\mathbf{x}) &= \bigtriangledown_{\mathbf{x}} \left[\mathbf{g}_f(\mathbf{x}) + \mathbf{J}_{\mathbf{h}}^T(\mathbf{x})\boldsymbol{\lambda} - \boldsymbol{\mu}\right] \\ &= \bigtriangledown_{\mathbf{x}} \mathbf{g}_f(\mathbf{x}) - \bigtriangledown_{\mathbf{x}} \left[\sum_{i=1}^{M} \lambda_i \frac{\partial h_i}{\partial x_1}, \cdots, \sum_{i=1}^{M} \lambda_i \frac{\partial h_i}{\partial x_N}\right]^T = \mathbf{H}_f(\mathbf{x}) - \sum_{i=1}^{M} \lambda_i \mathbf{H}_{h_i}(\mathbf{x}) \end{aligned} \tag{3.98}$$

is symmetric as the Hessian matrices $\mathbf{H}_f$ and $\mathbf{H}_{h_i}$, $(i = 1, \cdots, M)$ are symmetric.

We can now use the Newton-Raphson method to find the solution of the nonlinear equation $\mathbf{F}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}$ iteratively:

$$\begin{bmatrix} \mathbf{x}_{n+1} \\ \boldsymbol{\lambda}_{n+1} \\ \boldsymbol{\mu}_{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_n \\ \boldsymbol{\lambda}_n \\ \boldsymbol{\mu}_n \end{bmatrix} + \alpha \begin{bmatrix} \delta\mathbf{x}_n \\ \delta\boldsymbol{\lambda}_n \\ \delta\boldsymbol{\mu}_n \end{bmatrix} \tag{3.99}$$

where α is a parameter that controls the step size and $[\delta\mathbf{x}_n, \delta\boldsymbol{\lambda}_n, \delta\boldsymbol{\mu}_n]^T$ is the search direction (*Newton direction*), which can be found by solving the following equation:

$$\begin{aligned} \mathbf{J}_\mathbf{F}(\mathbf{x}_n, \boldsymbol{\lambda}_n, \boldsymbol{\mu}_n) \begin{bmatrix} \delta\mathbf{x}_n \\ \delta\boldsymbol{\lambda}_n \\ \delta\boldsymbol{\mu}_n \end{bmatrix} &= \begin{bmatrix} \mathbf{W}(\mathbf{x}_n) & \mathbf{J}_\mathbf{h}^T(\mathbf{x}_n) & -\mathbf{I} \\ \mathbf{J}_\mathbf{h}(\mathbf{x}_n) & \mathbf{0} & \mathbf{0} \\ \mathbf{M}_n & \mathbf{0} & \mathbf{X}_n \end{bmatrix} \begin{bmatrix} \delta\mathbf{x}_n \\ \delta\boldsymbol{\lambda}_n \\ \delta\boldsymbol{\mu}_n \end{bmatrix} \\ = -\mathbf{F}(\mathbf{x}_n, \boldsymbol{\lambda}_n, \boldsymbol{\mu}_n) &= - \begin{bmatrix} \mathbf{g}_f(\mathbf{x}_n) + \mathbf{J}_\mathbf{h}^T(\mathbf{x}_n)\boldsymbol{\lambda}_n - \mu_n \\ \mathbf{h}(\mathbf{x}_n) \\ \mathbf{X_n M_n 1} - \mathbf{1}/t \end{bmatrix} \end{aligned} \tag{3.100}$$

To get the initial values for the iteration, we first find any point inside the feasible region $\mathbf{x}_0$ and then $\boldsymbol{\mu}_0 = \mathbf{x}_0/t$, based on which we further find $\boldsymbol{\lambda}_0$ by solving the first equation in the KKT conditions in Eq. (3.96).

Actually this equation system above can be separated into two subsystems, which are easier to solve. Pre-multiplying $\mathbf{X}^{-1}$ to the third equation:

$$\mathbf{M}\delta\mathbf{x} + \mathbf{X}\delta\boldsymbol{\mu} = -\mathbf{XM1} + \mathbf{1}/t \tag{3.101}$$

we get

$$\mathbf{X}^{-1}\mathbf{M}\delta\mathbf{x} + \delta\boldsymbol{\mu} = -\mathbf{M1} + \mathbf{X}^{-1}\mathbf{1}/t = -\boldsymbol{\mu} + \mathbf{X}^{-1}\mathbf{1}/t \tag{3.102}$$

Adding this to the first equation:

$$\mathbf{W}\delta\mathbf{x} + \mathbf{J}_\mathbf{h}^T(\mathbf{x})\delta\boldsymbol{\lambda} - \delta\boldsymbol{\mu} = -\mathbf{g}_f(\mathbf{x}) - \mathbf{J}_\mathbf{h}^T(\mathbf{x})\boldsymbol{\lambda} + \boldsymbol{\mu} \tag{3.103}$$

we get

$$(\mathbf{W} + \mathbf{X}^{-1}\mathbf{M})\delta\mathbf{x} + \mathbf{J}_\mathbf{h}^T(\mathbf{x})\delta\boldsymbol{\lambda} = -\mathbf{g}_f(\mathbf{x}) - \mathbf{J}_\mathbf{h}^T(\mathbf{x})\boldsymbol{\lambda} + \mathbf{X}^{-1}\mathbf{1}/t = -\bigtriangledown_\mathbf{x} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \tag{3.104}$$

Combining this equation with the second one, we get an equation system of two vector variables $\delta\mathbf{x}$ and $\delta\boldsymbol{\lambda}$ with symmetric coefficient matrix:

$$\begin{bmatrix} \mathbf{W} + \mathbf{X}^{-1}\mathbf{M} & \mathbf{J}_\mathbf{h}^T(\mathbf{x}) \\ \mathbf{J}_\mathbf{h}(\mathbf{x}) & \mathbf{0} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x} \\ \delta\boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} \bigtriangledown_\mathbf{x} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) \\ \mathbf{h}(\mathbf{x}) \end{bmatrix} \tag{3.105}$$

Solving this equation system we get $\delta\mathbf{x}$ and $\delta\boldsymbol{\lambda}$, and we can further find $\delta\boldsymbol{\mu}$ by solving the third equation

$$\mathbf{M}\delta\mathbf{x} + \mathbf{X}\delta\mu = -\mathbf{XM1} - \mathbf{1}/t \tag{3.106}$$

to get

$$\delta\boldsymbol{\mu} = \mathbf{X}^{-1}\mathbf{1}/t - \mathbf{X}^{-1}\mathbf{M}\delta\mathbf{x} - \boldsymbol{\mu} \tag{3.107}$$

We now consider the two special cases of linear and quadratic programming:

- **Linear programming**:

$$\begin{array}{ll} \text{min:} & f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} \\ \text{s. t.:} & \mathbf{h}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}, \quad \mathbf{x} \geq \mathbf{0} \end{array} \tag{3.108}$$

We have
– $\mathbf{g}_f(\mathbf{x}) = \bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = \mathbf{c}$
– $\mathbf{J}_{\mathbf{h}}(\mathbf{x}) = \bigtriangledown_{\mathbf{x}}(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{A}$.
– $\mathbf{W}(\mathbf{x}) = \bigtriangledown^2_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{0}$.
The Lagrangian is

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{c}^T\mathbf{x} + \boldsymbol{\lambda}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) - \boldsymbol{\mu}^T\mathbf{x} \tag{3.109}$$

The KKT conditions are:

$$\begin{cases} \bigtriangledown_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{g}_f(\mathbf{x}) + \mathbf{J}^T_{\mathbf{h}}(\mathbf{x})\boldsymbol{\lambda} - \mu = \mathbf{c} + \mathbf{A}^T\boldsymbol{\lambda} - \boldsymbol{\mu} = \mathbf{0} \\ \mathbf{h}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0} \\ \mathbf{XM1} - \mathbf{1}/t = \mathbf{0} \end{cases}$$

The search direction of the iteration can be found by solving the this equation system:

$$\begin{bmatrix} \mathbf{0} & \mathbf{A}^T & -\mathbf{I} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{M} & \mathbf{0} & \mathbf{X} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x} \\ \delta\boldsymbol{\lambda} \\ \delta\boldsymbol{\mu} \end{bmatrix} = - \begin{bmatrix} \mathbf{c} + \mathbf{A}^T\boldsymbol{\lambda} - \boldsymbol{\mu} \\ \mathbf{A}\mathbf{x} - \mathbf{b} \\ \mathbf{XM1} - \mathbf{1}/t \end{bmatrix} \tag{3.110}$$

By solving the equation $\mathbf{c} + \mathbf{A}^T\boldsymbol{\lambda} - \boldsymbol{\mu} = \mathbf{0}$ we get the initial value $\boldsymbol{\lambda}$. The Matlab code for the interior point method for the LP problem is listed below:

```
function x=InterierPointLP(A,c,b,x)
    % given A,b,c of the LP problem and inital value of x,
    % find optimal solution x that minimizes c'*x
    [m n]=size(A);      % m constraints, n variables
    z1=zeros(n);
    z2=zeros(m);        % zero matrices in coefficient matrix
    z3=zeros(m,n);
    I=eye(n);           % identity matrix in coefficient matrix
    y=ones(n,1);
    t=9;                % initinal value for parameter t
    alpha=0.3;          % small enough not to exceed boundary
    mu=x./t;
    lambda=pinv(A')*(mu-c); % initial value of lambda
    w=[x; lambda; mu];      % initial values for all three
    B=[c+A'*lambda-mu; A*x-b; x.*mu-y/t];
    p=[x(1) x(2)];      % initial guess of solution
    while norm(B)>10^(-7)
```

```
        t=t*9;          % increase parameter t
        X=diag(x);
        M=diag(mu);
        C=[z1 A' -I; A z2 z3; M z3' X];
        B=[c+A'*lambda-mu; A*x-b; x.*mu-y/t];
        dw=-inv(C)*B;   % find search direction
        w=w+alpha*dw;   % step forward
        x=w(1:n);
        lambda=w(n+1:n+m);
        mu=w(n+m+1:length(w));
        p=[p; x(1), x(2)];
    end
    scatter(p(:,1),p(:,2));  % plot trajectory of solution
end
```

Example 3.5 A given linear programming problem shown below is first converted into the standard form:

$$\begin{array}{ll} \text{max:} & f(x_1,x_2)=2x_1+3x_2 \\ \text{s. t.:} & \left\{\begin{array}{l} 2x_1+x_2\le 18 \\ 6x_1+5x_2\le 60 \\ 2x_1+5x_2\le 40 \\ x_1\ge 0,\ \ x_2\ge 0 \end{array}\right. \end{array} \quad \Longrightarrow \quad \begin{array}{ll} \text{min:} & f(x_1,x_2,x_3,x_4,x_5)=-2x_1-3x_2 \\ \text{s. t,:} & \left\{\begin{array}{l} 2x_1+x_2+x_3=18 \\ 6x_1+5x_2+x_4=60 \\ 2x_1+5x_2+x_5=40 \\ x_i\ge 0,\ \ (i=1,\cdots,5) \end{array}\right. \end{array}$$

or in matrix form:

$$\begin{array}{ll} \text{min:} & f(\mathbf{x})=\mathbf{c}^T\mathbf{x} \\ \text{s. t. :} & \mathbf{h}(\mathbf{x})=\mathbf{A}\mathbf{x}-\mathbf{b}=\mathbf{0},\ \ \mathbf{x}\ge\mathbf{0} \end{array}$$

with

$$\mathbf{x}=\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix},\quad \mathbf{c}=\begin{bmatrix} -2 \\ -3 \\ 0 \\ 0 \\ 0 \end{bmatrix},\quad \mathbf{A}=\begin{bmatrix} 2 & 1 & 1 & 0 & 0 \\ 6 & 5 & 0 & 1 & 0 \\ 2 & 5 & 0 & 0 & 1 \end{bmatrix},\quad \mathbf{b}=\begin{bmatrix} 18 \\ 60 \\ 40 \end{bmatrix}$$

where x_3, x_4, x_5 are the slack variables.

We choose an initial value $\mathbf{x}_0=[1,\,2,\,1,\,1,\,1]^T$ and an initial parameter $t=9$, which is scaled up by a factor of 9 in each iteration, We also used full Newton step size of $\alpha=1$. As shown in Fig. 3.10, after 8 iterations, the algorithm converged to the optimal solution $\mathbf{x}^*=[5,\,6]^T$ corresponding to

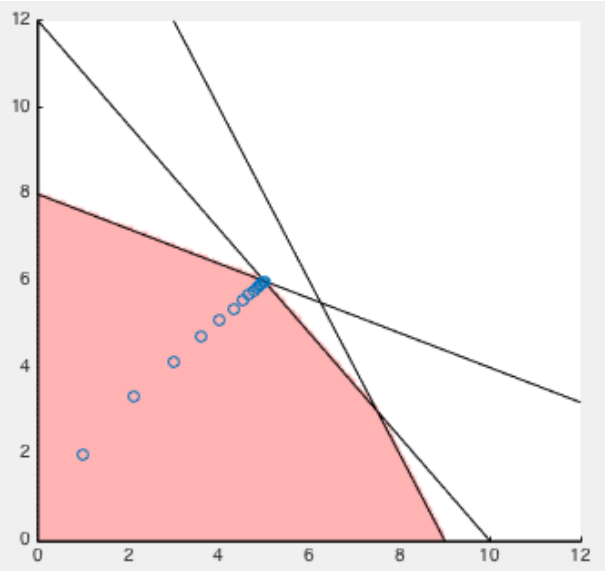


Figure 3.10 Interior Point Method for Linear Programming

the maximum function value $f(\mathbf{x}^*) = 2x_1 + 3x_2 = 28$:

	$(x_1$	$x_2)$	$f(\mathbf{x})$	error
1	$(1.000000e+00$	$2.000000e+00)$	-8.000000	52.413951
2	$(4.654514e+00$	$6.346354e+00)$	-28.348090	3.080204
3	$(5.040828e+00$	$5.946973e+00)$	-27.922575	0.213509
4	$(4.997282e+00$	$6.001764e+00)$	-27.999856	0.004262
5	$(4.999906e+00$	$5.999962e+00)$	-27.999697	0.000303
6	$(4.999989e+00$	$5.999996e+00)$	-27.999966	0.000034
7	$(4.999999e+00$	$6.000000e+00)$	-27.999996	0.000004
8	$(5.000000e+00$	$6.000000e+00)$	-28.000000	0.000000

At the end of the iteration, we also get the values of the slack variables: $x_3 = 2,\ x_4 = x_5 = 0$.

- **Quadratic programming:**

$$\begin{array}{ll} \text{min:} & f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ \text{s. t.:} & \mathbf{h}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}, \quad \mathbf{x} \geq \mathbf{0} \end{array} \tag{3.111}$$

We have

- $\mathbf{g}_f(\mathbf{x}) = \bigtriangledown_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q}\mathbf{x} + \mathbf{c}$
- $\mathbf{J}_{\mathbf{h}}(\mathbf{x}) = \bigtriangledown_{\mathbf{x}}(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{A}$.
- $\mathbf{W}(\mathbf{x}) = \bigtriangledown^2_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{Q}$.

The Lagrangian is

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} + \boldsymbol{\lambda}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) - \boldsymbol{\mu}^T\mathbf{x} \tag{3.112}$$

The KKT conditions are:

$$\begin{cases} \bigtriangledown_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{g}_f(\mathbf{x}) + \mathbf{J}_{\mathbf{h}}^T(\mathbf{x})\boldsymbol{\lambda} - \mu = \mathbf{Q}\mathbf{x} + \mathbf{c} + \mathbf{A}^T\boldsymbol{\lambda} - \boldsymbol{\mu} = \mathbf{0} \\ \mathbf{h}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0} \\ \mathbf{XM1} - \mathbf{1}/t = \mathbf{0} \end{cases}$$

The equation system for the Newton-Raphson method is:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{A}^T & -\mathbf{I} \\ \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{M} & \mathbf{0} & \mathbf{X} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x} \\ \delta\boldsymbol{\lambda} \\ \delta\boldsymbol{\mu} \end{bmatrix} = - \begin{bmatrix} \mathbf{Q}\mathbf{x} + \mathbf{c} + \mathbf{A}^T\boldsymbol{\lambda} - \boldsymbol{\mu} \\ \mathbf{A}\mathbf{x} - \mathbf{b} \\ \mathbf{XM1} - \mathbf{1}/t \end{bmatrix} \tag{3.113}$$

The Matlab code for the interior point method for the QP problem is listed below:

```
function [x mu lambda]=InteriorPointQP(Q,A,c,b,x)
    n=length(c);       % n variables
    m=length(b);       % m constraints
    z2=zeros(m);
    z3=zeros(m,n);
    I=eye(n);
    y=ones(n,1);
    t=9;               % initinal value for parameter t
    alpha=0.1;         % small stepsize not to exceed boundary
    mu=x./t;
    lambda=pinv(A')*(mu-c-Q*x); % initial value of lambda
    w=[x; lambda; mu];    % initial value
    B=[Q*x+c+A'*lambda-mu; A*x-b; x.*mu-y/t];
    while norm(B)>10^(-7)
        t=t*9;         & increase parameter t
        X=diag(x);
        M=diag(mu);
        C=[Q A' -I; A z2 z3; M z3' X];
        B=[Q*x+c+A'*lambda-mu; A*x-b; x.*mu-y/t];
        dw=-inv(C)*B;  % find search direction
        w=w+alpha*dw;  % step forward
        x=w(1:n);
        lambda=w(n+1:n+m);
        mu=w(n+m+1:length(w));
    end
end
```

Example 3.6 A 2-D quadratic programming problem shown below is to minimize four different quadratic functions under the same linear constraints as the

linear programming problem in the previous example. In general, the quadratic function is given in the matrix form:

$$\begin{aligned} &\text{min:} \quad f(\mathbf{x}) = [\mathbf{x}-\mathbf{m}]^T\mathbf{Q}[\mathbf{x}-\mathbf{m}] = \mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} + d \\ &\text{s. t.:} \quad \mathbf{h}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}, \;\; \mathbf{x} \geq \mathbf{0} \end{aligned}$$

with following four different sets of function parameters:

$$\mathbf{Q} = \begin{bmatrix} 4 & -1 \\ -1 & 4 \end{bmatrix}, \quad \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \quad \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \quad \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix}$$

$$\mathbf{m} = \begin{bmatrix} 4 \\ 10 \end{bmatrix}, \quad \begin{bmatrix} 9 \\ 5 \end{bmatrix}, \quad \begin{bmatrix} 7 \\ 2 \end{bmatrix}, \quad \begin{bmatrix} -1 \\ 6 \end{bmatrix}$$

As shown in the four panels in Fig. 3.11, the iteration of the interior point algorithm brings the solution from the same initial position at $\mathbf{x}_0 = [2,\ 1]^T$ to the final position for each of the four objective functions $f(\mathbf{x})$, the optimal solution, which is on the boundary of the feasible region in all cases except the third one, where the optimal solution is inside the feasible region, i.e., the optimization problem is not constrained. Also note that the trajectories of the solution during the iteration go straightly from the initial guess to the final optimal solution by following the negative gradient direction of the quadratic function in all cases except in the last one, where it makes a turn while approaching the vertical boundary, due obviously to the significantly greater value of the log barrier function close to the boundary, as shown in Fig. 3.9.

We also note that the dual problem in the SVM algorithm in Eq. (3.52) is in the same form as in Eq. (3.111), i.e., it is a quadratic programming problem that can be solved by the interior point algorithm.

Problems

1. Study carefully the first three sections of the chapter to understand the theoretical background of the constrained optimization problems. Then implement the algorithms discussed in the following sections for both linear and quadratic programming as stated in the following problems.
2. Develop Matlab code for the brute force method for solving an LP problem in the generic form:

$$\begin{aligned} &\text{maximize} && f(x_1, \cdots, x_N) = c_1x_1 + \cdots + c_Nx_N = \textstyle\sum_{i=1}^{N} c_ix_i \\ &\text{subject to:} && \begin{cases} a_{11}x_1 + \cdots + a_{1N}x_N = \sum_{i=1}^{n} a_{1i}x_i \leq b_1 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{M1}x_1 + \cdots + a_{MN}x_N = \sum_{i=1}^{N} a_{Mi}x_i \leq b_M \\ x_1 \geq 0, \cdots, x_N \geq 0 \end{cases} \end{aligned}$$

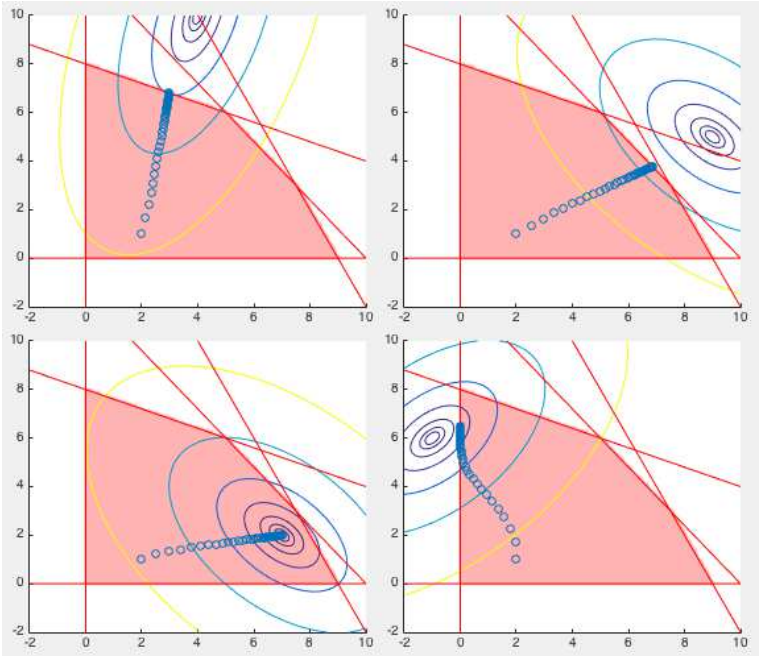


Figure 3.11 Interior Point Method for Quadratic Programming

Specifically, find all C_{N+M}^{N} intersections formed by $N+M$ hyperplanes in the N-D space corresponding to the M constraints $\mathbf{A}_{M\times N}\mathbf{x}_{N\times 1} \leq \mathbf{b}_{M\times 1}$ and the N conditions $\mathbf{x} \geq \mathbf{0}$, identify which of them are vertices of the polytope surrounded by these hyperplane, and find the vertex corresponding to the optimal solution that maximizes $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$ for the given $\mathbf{c}$.

Run your code to solve the LP problem in the example in Section 3.4.

3. Develop Matlab code to implement the simplex algorithm, then apply it to the LP problem in the previous problem, and the maximization problems:

$$\begin{aligned} &\text{maximize} && f(x_1, x_2, x_3) = 2x_1 - 3x_2 + 4x_3 \\ &\text{subject to:} && \begin{cases} 4x_1 - 3x_2 + x_3 \leq 3 \\ x_1 + x_2 + x_3 \leq 10 \\ 2x_1 + x_2 - x_3 \leq 10 \\ x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0 \end{cases} \end{aligned}$$

$$\begin{aligned} &\text{maximize} && f(x_1, x_2, x_3) = x_1 + 2x_2 - x_3 \\ &\text{subject to:} && \begin{cases} 2x_1 + x_2 + x_3 \leq 14 \\ 4x_1 + 2x_2 + 3x_3 \leq 28 \\ 2x_1 + 5x_2 + 5x_3 \leq 30 \\ x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0 \end{cases} \end{aligned}$$

4. Develop code to implement the interior point method, and apply it to both the LP and QP problems in the two examples in Section 3.7.

Part II

Regression

Equipped with the mathematical tools covered in Part I, we are now ready to discuss various machine learning (ML) methods and algorithms in the rest of the book, starting with Part II for regression analysis. The general learning process of all these learning methods may seem very different from each other in terms of their specific goals and mathematical setups, they all fit into a generic model that captures the essntial mechanism of the learning process.

In general all learning processes can be modeled as a black box, either a brain in human learning or a computer in machine learning, that takes some observed datasets as its input and generates the corresponding response as the output, representing some results learned from the input, depending on the specific learning task. As a simple ML example, the input data can be a set of numerical numbers, and the output a mathematical function that best characterizes certain relationship among these numbers. In a more complicated case, the input can be a set of images and the output some recognized objects that appear in the images. More generally, no matter how simple or complicated the learning problem is, the black box for the learning process can be represented symbolically in the simple form of a function $y = F(x, \theta)$, where the input x and output y of the model function may represent some large scale dataset such as images and text files as well as some numbers. Here θ denotes a set of parameters of the model function. In most ML methods, the specific form of the model function is presumed to be known or given, while the main task is to find the optimal parameters in such a way that the output represents the best results learned from the data in some sense.

All learning methods covered in the book fit into the generic black box model. In Part II for regression analysis, the model function is simply a regression function $y = f(x, \theta)$ that characterizes the relationship between two sets of observed sample values of the input and output variables x and y. In Part IV, the model functions of the various classification methods are some specific algorithms (naive Bayes, support vector machine, etc.), and the goal of the learning process is to find the parameters used in the algorithm that best fit the given data. In Part V discussing the general method of artificial neural networks (ANN), the model functions are always a network composed of two or more layers of neurons or nodes, and the parameters are the connectivities or weights between the nodes of any two neighboring layers. While the number of model parameters in all these learning methods may vary tremendously, from two in the linear regression function $y = f(x, \theta) = ax + b$, to many millions of weights in some deep learning neural networks or even billions of weights in some Large language models, they all fit into the black box learning model with the common goal of determining the model parameters represented by θ. Here the model parameters are not to be confused with some *hyperparameters* in many ML algorithms, which, different from the model parameters, may be used to control the learning process, such as the size of an ANN, or how closely the model needs to fit the observed data contaminated with noise.

In Part II, we will consider various methods for regression analysis, which is a process of *supervised learning*, with the general goal of establishing a model

for the relationship between a set of independent varialbes (stimuli) and another set of dependent variables (responses) of a generic system. Such a model can be symbolically represented as $y = f(\mathbf{x}, \boldsymbol{\theta})$, where $\mathbf{x} = [x_1, \cdots, x_d]^T$ is a column vector containing d independent variables, y is a scalar for the dependent variable, and column vector $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_M]^T$ contains a set of M parameters of the model. The main task of a supervised learning process is to obtain the unknown model parameters in $\boldsymbol{\theta}$ by learning from a set of observed data, called the *training set*, represented by $\mathcal{D} = \{(\mathbf{x}_n, y_n),\ (n = 1, \cdots, N)\}$, containing a set N data samples $\mathbf{x}_n$ each labeled in some way by the corresponding y_n.

Specially, in regression, as one of the supervised learning methods, the model $y = f(\mathbf{x}, \boldsymbol{\theta})$ is a specific *regression function* that maps a data point $\mathbf{x}$, a point in the d-dimensional space, to a scalar output y, represented by a hypersurface in a $d + 1$ dimensional space spanned by y as well as the d independent variables in $\mathbf{x}$. In particular, the hypersurface is a curve if if $d = 1$, or a surface if $d = 2$.

The regression function can be assumed to be either deterministic or probabilistic depending on the specific algorithm. Typically the regression function is hypothesized to take certain form (e.g., linear, higher order polynomial, or exponential) based on some prior knowledge, while the parameters in $\boldsymbol{\theta}$ are to be determined by the regression algorithm based on the training set $\mathcal{D}$, which can also be denoted by $\mathcal{D} = \{\mathbf{X},\ \mathbf{y}\}$ in terms of a $d \times N$ matrix $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ and an N-D vector $\mathbf{y} = [y_1, \cdots, y_N]^T$. Through out the discussion in all the following chapters, we will always assume the data samples in the training set are *independently and identically distributed (i.i.d.)*.

The goal of regression is to obtain the parameters in $\boldsymbol{\theta}$ so that the model prediction $\hat{y}_n = f(\mathbf{x}_n, \boldsymbol{\theta})$ matches optimally the corresponding y_n as the ground truth, i.e., the hypersurface of the model best fits all training samples in the $d+1$ dimensional space.

In this part, we will first in Chapter 4 treat regression as a simple and typical supervised learning process and illustrate some important issues such as underfitting versus overfitting commonly faced in general by all supervised learning methods including not only regression algorithms considered in this Part, but also the classification algorithms to be considered in Parts IV and V. We will consider first various linear and nonlinear regression methods based on deterministic models in Chapters 5 and 6, and then some additional regression methods based on probabilistic models in Chapters 7 and 8, which can also be applied to classification problems.

4 Bias-Variance Tradeoff and Overfitting vs Underfitting

Before discussing various regression algorithms, we first in this chapter consider some guiding philosophy and methodology of supervised learning in general, such as frequentist versus Bayesian approach; the tradeoff and balance between bias and variance errors, or similarly, between overfitting and underfitting of the data; and some general approaches commonly used to address these issues, such as cross-validation and ensemble learning. Although here we consider such issues mostly in the context of regression analysis, we will come back to them over and over again when we discuss other supervised learning algorithms through out the book, all of which need to address the same general issues by the same general approaches, each in its own specific way.

4.1 Supervised Learning as Optimization

In general, supervised learning, including both regression and classification as the two typical types of problems, can be formulated as a system modeling process based on the given training set $\mathcal{D} = \{\mathbf{X},\ \mathbf{y}\}$, when the independent variable $\mathbf{x} \in \mathbf{X}$ and the dependent variable $y \in \mathbf{y}$ are treated respectively as the input (stimuli) and output (responses) of a system, of which the behavior in terms of the relationship between such input and output is to be modeled by a generic function $y = f(\mathbf{x}, \boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta}$).

In general, a supervised learning problem can be formulated as the framework illustrated in Fig. 4.1, with the goal of constructing a mathematical model, symbolized by function $f(\mathbf{x}, \boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta}$, in such a way that its output $\hat{\mathbf{y}}$ matches the labeling y corresponding to the input $\mathbf{x}$ in the training set $\mathcal{D}$ in some sense. Typically the form of the model $f(\mathbf{x}, \boldsymbol{\theta})$ is assumed to be known, which is either a specific function in regression, or certain specific classification algorithm (e.g., statistic model or a neural network) in classification). In any case, the model parameters in $\boldsymbol{\theta}$ are to be determined optimally in some sense, such as the discrepancy between the actual output $\hat{\mathbf{y}}$, the prediction of the model, and the desired output $\mathbf{y}$, the ground truth given in the training set is minimized. In other words, a learning problem is typically solved as an optimization problem.

Specifically, in regression analysis, the model $y = f(\mathbf{x}, \boldsymbol{\theta})$ is simply a function with independent variables $\mathbf{x}$ and dependent variable y, while in pattern

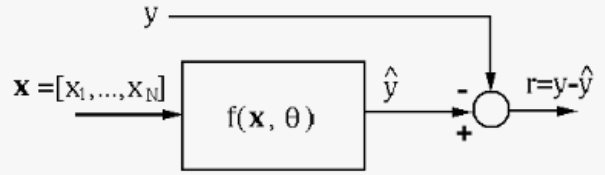


Figure 4.1 A General Model for Supervised Learning

classification (or object recognition) (to be discussed in Part IV), the model $y = f(\mathbf{x}, \boldsymbol{\theta})$ represents a specific classification method, by which a pattern described by the independent variable $\mathbf{x}$ as its *features* is assigned to one of the classes $\{C_1, \cdots, C_K\}$ indicated by the corresponding dependent variable y. As both regression and classification can be considered as a general supervised learning process, much of the discussion in this chapter for various regression algorithms is also highly relevant to the classification algorithms to be considered in the future chapters.

The goal of a regression algorithm is for the regression model to optimally fit the observed data, so that ideally it reflects all variations due to the valid *signal* in the data, but at the same time remains unaffected by the inevitable observation *noise* also existing in the data. In general, all learning algorithms aiming at modeling a set of observed data face the common dilemma between over and under trusting the observed data contaminated by noise, and they all need to address the same issue of how to best distinguish between signal and noise in the data, so that they neither *overfit* the data, i.e., overly affected by noise, nor *underfit* data, i.e., not matching the valid signal variation sufficiently closely. This issue of making a proper tradeoff between overfitting and underfitting will need to be addressed by various means in different learning algorithms, as we will see in this and future chapters.

The problem of regression analysis can be addressed from different philosophical viewpoints. First, from the *frequentist* point of view, the unknown model parameters in $\boldsymbol{\theta}$ are fixed deterministic variables and their values are to be estimated based on the training set. A typical method based on this viewpoint is the *least squares (LS)* method for both linear and nonlinear regression methods. Alternatively, from the point of view of *Bayesian inference*, the model parameters in $\boldsymbol{\theta}$ are random variables. We can estimate their *prior probability distribution* $p(\boldsymbol{\theta})$ based on some prior knowledge without observing any data. If no such prior knowledge is available, we can simply assumed $p(\boldsymbol{\theta})$ is a uniform distribution, i.e., all possible values of $\boldsymbol{\theta}$ are equally likely. Once the training data $\mathbf{X}$ are available and observed, we can estimate the *posterior probability* of the model parameters based on Bayes' theorem:

$$p(\boldsymbol{\theta}|\mathbf{X}) = \frac{p(\mathbf{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{X})} \propto p(\mathbf{X}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{4.1}$$

In general, the posterior $p(\boldsymbol{\theta}|\mathbf{X})$ is a narrower distribution than the prior $p(\boldsymbol{\theta})$, and therefore a more accurate description of the model parameters, due to the reduced uncertainty of the parameters based on additional information gained from the observed data. The output $\hat{y} = f(\mathbf{x}, \boldsymbol{\theta})$ of such a regression model based Bayesian inference is no longer a deterministic value, but a random variable described by its probability distribution in terms of its mean and variance. Typical methods based on this point of view include Bayesian linear regression, logistic regression and Gaussian process regression.

Based on either of these two viewpoints, different regression methods can be used to find the parameters $\boldsymbol{\theta}$ for the model function $\hat{y} = f(\mathbf{x}, \boldsymbol{\theta})$, such as those listed below.

- *Least squares (LS) method*:

  This frequentist method measures how well the regression function models the training data by the *residual* defined as $r(\boldsymbol{\theta}) = y - \hat{y}(\boldsymbol{\theta})$, defined as the difference between the model prediction $\hat{y}(\boldsymbol{\theta}) = f(\mathbf{x}, \boldsymbol{\theta})$ and the ground truth labeling value y corresponding to $\mathbf{x}$, for each of the N data points in the observed data:

$$\mathbf{r}(\boldsymbol{\theta}) = \begin{bmatrix} r_1 \\ \vdots \\ r_N \end{bmatrix} = \begin{bmatrix} y_1 - \hat{y}_1 \\ \vdots \\ y_N - \hat{y}_N \end{bmatrix} = \begin{bmatrix} y_1 - f(\mathbf{x}_1, \boldsymbol{\theta}) \\ \vdots \\ y_N - f(\mathbf{x}_N, \boldsymbol{\theta}) \end{bmatrix} = \mathbf{y} - \hat{\mathbf{y}} \tag{4.2}$$

  Ideally, for the residual to be zero, $\mathbf{r}(\boldsymbol{\theta}) = \mathbf{0}$, we find the M model parameters as components of $\boldsymbol{\theta}$ by solving this nonlinear equation system of N equations of M unknowns in $\boldsymbol{\theta}$. However, as in general $N \gg M$, this system is overconstrained without a solution. We therefore can only use least squares method to find the optimal solution that minimizes the *sum of squared error (SSE)*:

$$\varepsilon(\theta) = \frac{1}{2}||\mathbf{r}||^2 = \frac{1}{2}\mathbf{r}^T\mathbf{r} = \frac{1}{2}\sum_{n=1}^{N} r_n^2 = \frac{1}{2}\sum_{n=1}^{N}(y_n - f(\mathbf{x}_n, \boldsymbol{\theta}))^2 \tag{4.3}$$

  Here the coefficient $1/2$ is included for mathematical convenience. When divided by the total number of samples N, this SSE becomes the average of all N squared errors, and the error becomes the *mean squared error (MSE)*. Minimizing either SSE or MSE produces the same results.

- *Maximum likelihood estimate (MLE):*

  This Bayesian inference method measures how well the regression function models the training data in terms of the likelihood $L(\boldsymbol{\theta}|\mathcal{D})$ of the model parameter $\boldsymbol{\theta}$ based on the observed dataset $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$, which is proportional to the conditional probability of $\mathcal{D}$ given $\boldsymbol{\theta}$:

$$L(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta}) = p(\mathbf{y}|\mathbf{X}, \theta) \tag{4.4}$$

  The optimal model parameters in $\boldsymbol{\theta}$ can be found as those that maximize $L(\boldsymbol{\theta}|\mathcal{D})$ or equivalently $p(\mathbf{y}|\mathbf{X}, \theta)$.

We assume the residual $r = y - f(\mathbf{x}, \boldsymbol{\theta})$ is a zero-mean random variable (without loss of generality) with a normal probability density function (pdf):

$$\begin{aligned} p(r) = p(y - f(\mathbf{x}, \boldsymbol{\theta})) &= \mathcal{N}(r, 0, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - f(\mathbf{x}, \boldsymbol{\theta}))^2}{2\sigma^2}\right) \\ &= \mathcal{N}(y, f(\mathbf{x}, \boldsymbol{\theta}), \sigma^2) = p(y|\mathbf{x}, \boldsymbol{\theta}) \end{aligned} \tag{4.5}$$

The justification for this assumed normal pdf is that it has the greatest entropy (Section B.2.1), or maximum uncertainty among all possible pdfs with the same variance, i.e., given σ^2, the assumed normal pdf imposes least amount of constraint and thereby minimum bias.

The zero-mean pdf of $r = y - f(\mathbf{x}, \boldsymbol{\theta})$ can also be considered as the pdf of y with mean $f(\mathbf{x}, \boldsymbol{\theta})$, the conditional pdf $p(y|\mathbf{x}, \boldsymbol{\theta})$ of y given $\boldsymbol{\theta}$ as well as $\mathbf{x}$, based on the assumption that y and $\mathbf{x}$ are related by $y = f(\mathbf{x}, \boldsymbol{\theta})$. Then we can find the likelihood of $\boldsymbol{\theta}$ given the data samples in the training set, all assumed to be independently and identically distributed:

$$\begin{aligned} L(\theta|\mathcal{D}) = p(\mathbf{y}|\mathbf{X}, \theta) &= \prod_{n=1}^{N} p(y = y_n|\mathbf{x}_n, \theta) \\ &= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_n - f(\mathbf{x}_n, \boldsymbol{\theta}))^2}{2\sigma^2}\right) \end{aligned} \tag{4.6}$$

or, equivalently and for mathematical convenience, minimize the negative log likelihood function:

$$-l(\boldsymbol{\theta}|\mathcal{D}) = -\log L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y}) = -\sum_{n=1}^{N} \log p(y = y_n|\mathbf{x}_n, \theta) \tag{4.7}$$

$$= -\sum_{n=1}^{N} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_n - f(\mathbf{x}, \boldsymbol{\theta}))^2}{2\sigma^2}\right)\right]$$

$$= \frac{N}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{n=1}^{N} ||\mathbf{r}||^2 \tag{4.8}$$

The first term and the coefficient $1/\sigma^2$ in the second term are both independent of θ and can therefore be dropped. We note that this result is equivalent to that in Eq. (4.3), i.e., we can find the optimal $\boldsymbol{\theta}$ by either maximizing the likelihood $L(\boldsymbol{\theta}|\mathcal{D})$, or equivalently minimizing the SSE $\varepsilon(\boldsymbol{\theta})$.

We further note that negative log likelihood $-l(\boldsymbol{\theta}|\mathcal{D}) = -\log L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y})$ in the equation above can also be interpreted differently as the cross entropy (Section B.2.2) between an unknown ground truth probability P and the model probability Q:

$$H(P, Q) = -\sum_{i} P_i \log Q_i = -E_P[\log Q] \tag{4.9}$$

We rewrite Eq. (4.7) as the average of the logarithm of the model distribution $Q = p(y|\mathbf{x}, \theta)$ over all N observed samples in the dataset $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$:

$$-l(\boldsymbol{\theta}|\mathcal{D}) = -\sum_{n=1}^{N} \log p(y = y_n|\mathbf{x}_n, \theta) = -E_{(X,y)} \left[\log p(y = y_n|\mathbf{x}_n, \theta)\right] \tag{4.10}$$

and realize that this is the cross entropy $H(P, Q)$ estimated based on the N samples, which can be minimized for model Q to match the ground truth P as closely as possible.

- *Maximum A Posteriori (MAP)*

  This is also a Bayesian inference method that measures how well the regression function models the training data by the posterior probability of the parameter $\boldsymbol{\theta}$ given in Eq. (4.1), proportional to the product of the likelihood $L(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta})$ and the prior $p(\boldsymbol{\theta})$. If no prior knowledge about $\boldsymbol{\theta}$ is available, then $p(\boldsymbol{\theta})$ is a uniform distribution, then MAP is equivalent to MLE. However, if certain prior knowledge regarding $\boldsymbol{\theta}$ does exist, and the prior is not uniform, then the posterior better characterizes $\boldsymbol{\theta}$ than the likelihood and MAP may produce better result than MLE.

We conclude that regression analysis, and in general many supervised learning algorithms to be considered in the following chapters, can be treated as an over-constrained optimization problem with many more observed data points in the training set than the number of unknown parameters in the model. Following different philosophical view points, this optimization problem can be addressed differently, such as minimizing the sum of squared errors, maximizing the likelihood of the model parameters, or minimizing the negative likelihood, i.e., the cross entropy between the model and unknown ground truth distributions of the given dataset, to be discussed in details in the following sections.

More generally, some of the concepts and methods in regression analysis discussed in the following sections are also shared by many supervised learning algorithms for classification. For example, the general method of neural network is conceptually based on some ideas very different from regression, but both of these two methods can be described mathematically as a function $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\theta})$ that models the relationship between the independent variable $\mathbf{x}$ and dependent variable $\mathbf{y}$. While this function is called the regress function in this chapter, it is also a way to describe the complex behavior of a multilayer learning network to be discussed in Section 18.2

4.2 Bias-Variance Tradeoff

Before discussing specifically various regression algorithms, we first address an important issue that commonly appears in not only regression problems as discussed below, but also in general all supervised learning problems based on training dataset (or simply training set) such as classification.

In a generic regression problem, the goal is to model the unknown relationship between a dependent variable $y = f(\mathbf{x})$ and d independent variables in $\mathbf{x} = [x_1, \cdots, x_d]^T$, by a hypothesis regression function $\hat{f}(\mathbf{x})$, based on a set of training data $\mathcal{D} = \{(\mathbf{x}_n, y_n)\ n = 1, \cdots, N\}$ containing N observed data samples. As the observation is inevitably contaminated by some random noise e, both the observed value modeled by $y = f(\mathbf{x}) + e$ and the regression model $\hat{f}(\mathbf{x})$ need to be treated as random variables. We need to consider all possible errors of different types in the regression analysis as the consequence of such noisy data.

We first assume without loss of generality that the random noise e has a zero mean $E[e] = 0$ and variance σ^2, and it is independent of both f and $\hat{f}$:

$$\begin{aligned} &\text{Var}[e] = E[(e - E[e])^2] = E[e^2] = \sigma^2 \\ &E[e\, f] = E[e]\ E[f], \quad E[e\, \hat{f}] = E[e]\ E[\hat{f}] \end{aligned} \tag{4.11}$$

To measure how well the model $\hat{f}(\mathbf{x})$ fits the noisy data $y = f(\mathbf{x}) + e$, we define the *mean squared error (MSE)* between them (with $\mathbf{x}$ dropped for simplicity):

$$\begin{aligned} \text{E}[(y - \hat{f})^2] &= \text{E}[(f + e - \hat{f})^2] = \text{E}\left[[(f - E\hat{f}) + (E\hat{f} - \hat{f}) + e]^2\right] \\ &= E\left[(f - E\hat{f})^2 + (E\hat{f} - \hat{f})^2 + e^2 + 2e(f - E\hat{f}) + 2e(E\hat{f} - \hat{f}) + 2(f - E\hat{f})(E\hat{f} - \hat{f})\right] \\ &= \text{E}[(f - E\hat{f})^2] + \text{E}[(E\hat{f} - \hat{f})^2] + E(e^2) \\ &\quad +2\,\text{E}[e(f - E\hat{f})] + 2\,\text{E}[e(E\hat{f} - \hat{f})] + 2\,\text{E}[(E\hat{f} - \hat{f})(f - E\hat{f})] \\ &= \text{E}[(f - E\hat{f})^2] + \text{E}[(E\hat{f} - \hat{f})^2] + \text{E}[e^2] \end{aligned} \tag{4.12}$$

The last equality is due to the fact that all three cross terms are zero:

$$\begin{cases} \text{E}[e(f - E\hat{f})] = \text{E}[e]\ \text{E}[f - E\hat{f}] = 0 \\ \text{E}[e(E\hat{f} - \hat{f})] = \text{E}[e]\ \text{E}[E\hat{f} - \hat{f}] = 0 \\ \text{E}[(E\hat{f} - \hat{f})(f - E\hat{f})] = (E\hat{f} - E\hat{f})(f - E\hat{f}) = 0 \end{cases} \tag{4.13}$$

Now the total mean squared error in Eq. (4.12) can be expressed in terms of three different types of errors:

$$\text{Mean Squared Error} = \text{Variance Error} + \text{Bias Error} + \text{Irreducible Error} \tag{4.14}$$

where

- *Bias Error:* $\text{E}[(f - E\hat{f})^2] = (f - E\hat{f})^2$ measures how well the estimated function $\hat{f}$ models the ground truth function f;
- *Variance Error:* $\text{E}[\hat{f} - E\hat{f}]^2 = \text{Var}[\hat{f}]$ measures the variation of the model;
- *Irreducible Error* $\text{E}[e^2] = \sigma^2$ measures the inevitable observation noise.

The variance error and bias error are illustrated in Fig. 4.2. Specially comparing the top right and the bottom left cases we see that under the same total MSE and irreducible error σ^2, the bias error and the variance error are complementary to each other, i.e., if one is higher, the other is lower, depending on the specific model function $\hat{f}$. This relationship is called the *bias-variance tradeoff*, which is related to the common issue of overfitting versus underfitting in all

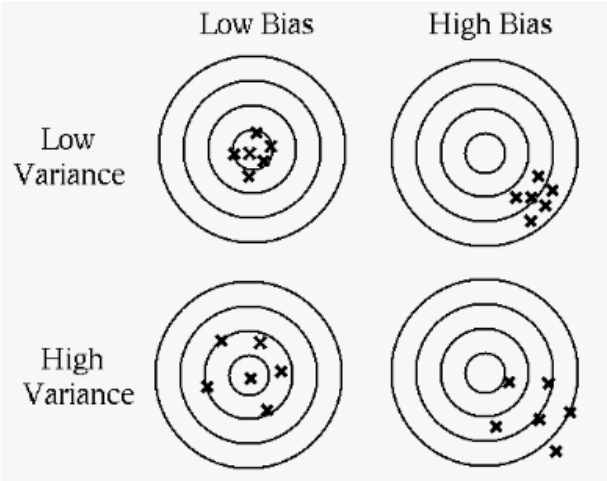


Figure 4.2 Bias-Variance Tradeoff

supervised learning problems in general as we will see in many of the learning algorithms through out the book, such as support vector machine in Chapter 14, back-propagation network in Chapter 18, and reinforcement learning in Chapter 20.

- *Overfitting:*
  To match the training data closely, a complex model may be needed to take into consideration all detailed variations in the data which may be caused by the noise in the data. Such a model tends to have a low variance error, but may suffer from high bias error, as it may be overly affected by the random noise such as some outliers, as shown in the top-right case in Fig. 4.2. Such a model does not generalize well across multiple datasets contaminated differently by noise.
- *Underfitting:*
  On the other hand, to avoid being overly affected by the noise, a simple model may be used with the possible consequence that it may not match the detailed variation due to the valid signal in the data sufficiently closely. Such a model tends to have low bias error as it is insensitive to the varying noise across datasets, but it may suffer from high variance error as it may not match all valid signal variations. This is the lower-left case in Fig. 4.2.

	Overfit	Underfit
Variance error	High	Low
Bias error	Low	High

(4.15)

These two possibilities are illustrated by the examples in Fig. 4.3.
As both overfitting and underfitting result in poor prediction capability, they

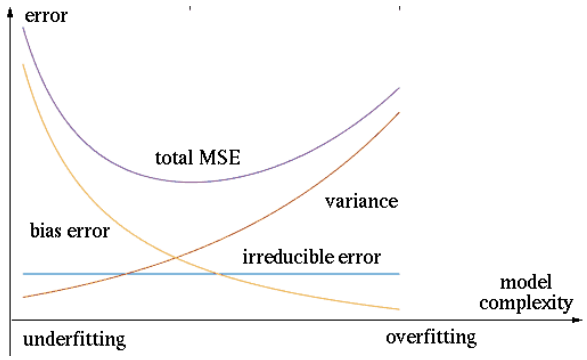


Figure 4.3 Under vs. Overfitting

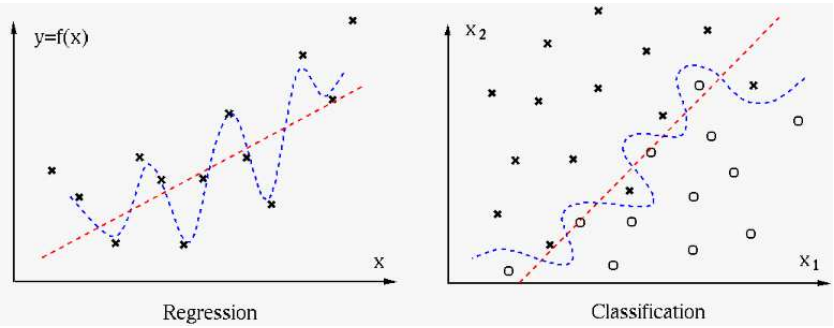


Figure 4.4 Under and Overfitting in Regression and Classification

need to be avoided by making a proper tradeoff between bias-error/underfitting and variance-error/overfitting. This is one of the most important issues in general in all supervised learning algorithms aiming at modeling a system and predicting its behaviors, including regression for predicting the output value $\hat{y} = f(\mathbf{x})$ as a function of a given data point $\mathbf{x}$, and classification for providing a categorical class labeling of a given point $\mathbf{x}$.

The issue of overfitting versus underfitting is further illustrated in Fig. 4.4. In the case of regression (left panel), the data points are modeled by two different regression functions, and in classification, the 2-D space is partitioned into two regions corresponding to two classes by two different *decision boundaries*. In both cases, the given datasets are overfitted by the blue curve underfitted by the red curve.

- Underfitting:

  The simple linear regression function or decision boundary (red) may underfit the data as some legitimate variations in the data may be missed. This is a problem if all data points reliably reflect some complex variations due to valid signal in the data.

- Overfitting:

  The complex regression model (e.g., a high order polynomial) or decision boundary (blue) may overfit the data if it is heavily affected by the variation due to observation noise instead of valid signal in the data. This is a problem if the data samples are taken from a simple process, but are fitted by an unnecessarily complex model due to the noise, i.e., a violation of the principle of Occam's razor.

Therefore we face the challenge of how to make a proper tradeoff between over and underfittings, so that the algorithm can produce an optimal model that fits the variations in the given data, in the sense that it maximally fits the valid signal in the data, but at the same time minimally affected by the noise also in the data. Various methods to address this challenge will be considered in the following section and through out the discussion of many supervised learning algorithms in the following chapters.

4.3 Cross-Validation

In the case when the entire training set is used for training to take full advantage of the available data which may be limited to start with, it is impossible to distinguish between variations due to valid signal and those due to random observation noise, and to judge whether the model produced by the learning algorithm in question either over or underfits the given data. This problem can be addressed by the method of cross-validation. If the dataset available for training the algorithm is of sufficient size, it can be partitioned into two or more subsets each used to either train or test the algorithm. This method is based on the simple idea that the valid signal variation should be shared by all sub-datasets, while they are contaminated differently by the noise due to its random nature. If the algorithm is trained by one subset of data but tested by another, then its performance measured by certain error (calculated differently depending on the specific learning problem and algorithm used, e.g., residual for regression, or rate of error for classification) can be used to assess the algorithm in terms of whether it suffers from the overfitting problem.

Cross-validation has different variations depending on how the available dataset is partitioned.

- Exhaustive methods: the algorithm is trained and tested based on the available dataset divided in all possible ways into multiple subsets. Typically, the leave-p-out cross-validation uses p out of the total n sample points in the dataset to test the algorithm after it is trained by the remaining $n - p$ samples. To achieve unbiased result, we would need to repeat this process C_p^n times to exhaust all possible ways to choose p samples out of n. To avoid repeating too many times when n is large, we may need to consider the special case of either leave-pair-out ($p = 2$) or leave-one-out ($p = 1$).

- Non-exhaustive methods: Typically the K-fold cross-validation divides the dataset into K (e.g., $K = 4$ or 5) equal-sized subsets, of which $K - 1$ subsets are used for training while the remaining one is reserved for testing. This process is repeated K times each based on a different testing subset and then the average of the K errors is used to check if the algorithm suffers overfitting. Specially we can simply let $K = 2$, or we can also randomly divide the available data into two subsets each for either training or testing, and calculate the average of the errors obtained while the process is repeated multiple times.

The method of cross-validation method can be used to detect and assess the algorithm in question to see whether or how much it over or under fits the given data, it has certain limitations. First, the method requires much increased computational complexity. Second, the method is effective only when the size of the training set available is sufficiently large and can therefore be divided into multiple subsets of meaningful sizes. If the size of the available dataset if small, as may be the case in some applications, the results generated by the cross-validation may not be reliable and meaningful.

4.4 Regularization and Ensemble Learning

The method of cross-validation can detect and assess overfitting problems, but it does not correct the problem by itself. Some additional approaches can be considered to address this problem.

First, we consider *regularization*, which is an method widely used in many supervised learning algorithms (including both regression and classification) to deal with the issue of overfitting versus underfitting. Typically this is done by adjusting certain hyperparameters of the model to make a proper tradeoff between the two extremes of over and underfittings. In particular, if the problem at hand is ill-conditioned (see Section A.8.5), then regularization can be highly effective so that the algorithm is less sensitive to noise in the dataset and therefore less overfitting, i.e., it is more generalizable while still able to capture the essential nature and behavior of the valid signal. We will show how regularization is actually implemented, such as the ridge regression in Section 5.2, and many other learning algorithm to be considered in future chapters.

Second, we consider the general method of *Ensemble learning*, a general approach that take advantage of multiple learning algorithms, each considered as a hypothesis, trained on multiple datasets, so that the resulting model can best fits the give data. This method is based on the idea that a single learning algorithm trained on a single dataset, here called a learner, may be *weak* as it may suffer from the problem of under vs overfitting, but a set of multiple such weak learners (either identical or different) can be trained on multiple datasets and combined in some way to become a *strong* learner. In general such a ensemble

learning method comes with two flavors depending on how the weak learners are combined to form a strong learner.

- *Bootstrap Aggregation (Bagging)*

  This is a two-step process including *bootstrapping* in which a set of independent weak learners are trained in parallel by different subsets as random subsamples from the total dataset (with replacement), and *aggregation* in which a strong learner combines the results by the individual weak learners with equal weights, such as their average for regression, or their majority vote for classification. The weak learners can be either identical (homogeneous) or different (heterogeneous). In the latter case, the method is also called *stacking*, as the collection of all such weak learners based on different algorithms and learning in parallel are treated as a stack. As we will see in Section 16, the popular method of *random forest* is a bagging algorithm that combines multiple decision tree classifiers for better classification performance.
- *Boosting*

  This is a multi-step sequential process in which the weak learners are trained sequentially, in the sense that each learner is trained on different subset of data, based on the learner in the previous step, thereby continuously improving the overall performance of the strong learner. As we will see in Section 13.4, the method of *adaptive boosting or AdaBoost* is such a boosting algorithm for classification.

5 Linear Regression

5.1 Linear Least Squares (LLS) Regression

In linear regression, the relationship between the dependent variable y and the d independent variables $\{x_1, \cdots, x_d\}$ is modeled by a linear hypothesis function:

$$\hat{y} = f(\mathbf{x}, \boldsymbol{\theta}) = f(\mathbf{x}, \mathbf{w}, b) = b + \sum_{i=1}^{d} w_i x_i = b + \mathbf{x}^T \mathbf{w} \tag{5.1}$$

parameterized by $\boldsymbol{\theta}$ containing b and $\mathbf{w} = [w_1, \cdots, w_d]^T$ for the coefficients or weights in the linear function. The goal of a linear regression algorithm is to estimated the model parameters in $\boldsymbol{\theta}$ basied on the training set $\mathcal{D} = \{\mathbf{X},\ \mathbf{y}\}$ containing a set of N given data points in $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ and the corresponding labelings in $\mathbf{y} = [y_1, \cdots, y_N]^T$.

For notational convenience, we redefine both $\mathbf{x}$ and $\mathbf{w}$ so that they become augmented $d+1$ dimensional vectors: $\mathbf{x} = [x_0, x_1, \cdots, x_d]^T$ with an extra component $x_0 = 1$ and $\mathbf{w} = [w_0, w_1, \cdots, w_d]^T$ with an extra component $w_0 = b$. Now the linear model above can be expressed more concisely as

$$\hat{y} = f(\mathbf{x}, \mathbf{w}) = \mathbf{x}^T \mathbf{w} \tag{5.2}$$

Geometrically, the linear regression function $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$ is a straight line if $d = 1$, a plane if $d = 2$, or a hyperplane if $d > 2$, in the $d+1$ dimensional space spanned by y as well as $\{x_1, \cdots, x_d\}$. The function has an intercept $f(\mathbf{0}) = w_0 = b$ along the y dimension (when $x_1 = \cdots = x_d = 0$), and a normal vector $[w_1, \cdots, w_d, -1]^T$. When the regression function is set to zero, i.e., the regression line or plane is thresholded by the plane $y = 0$, it becomes an equation $f(\mathbf{x}) = \mathbf{x}^T \mathbf{w} = 0$, representing a point if $d = 1$, a straight line if $d = 2$, a plane or hyperplane if $d \geq 3$, in the d-dimensional space spanned by $\{x_1, \cdots, x_d\}$, as shown in Fig. 5.1 for $d = 2$.

We denote by $\mathbf{X}$ the matrix containing the N augmented data points $\mathbf{x}_n = [x_{0n} = 1, x_{1n}, \cdots, x_{dn}]^T$ $(n = 1, \cdots, N)$ as its column vectors:

$$\mathbf{X} = \begin{bmatrix} 1 & \cdots & 1 \\ x_{11} & \cdots & x_{1N} \\ \vdots & \ddots & \vdots \\ x_{d1} & \cdots & x_{dN} \end{bmatrix}_{(d+1)\times N} = [\mathbf{x}_1, \cdots, \mathbf{x}_N] \tag{5.3}$$

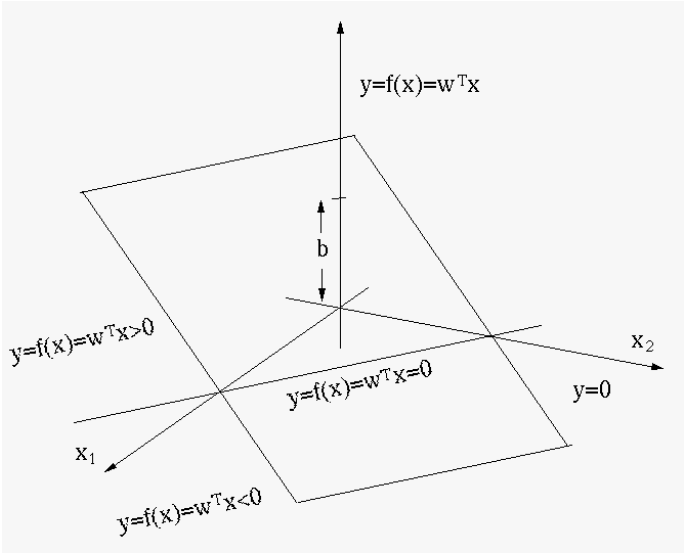


Figure 5.1 Linear Regression as a Binary Classifier

and its transpose can be written as:

$$\mathbf{X}^T = \begin{bmatrix} 1 & x_{11} & \cdots & x_{d1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{1N} & \cdots & x_{dN} \end{bmatrix}_{N\times(d+1)} = [\mathbf{1}, \underline{\mathbf{x}}_1, \cdots, \underline{\mathbf{x}}_d] \tag{5.4}$$

where $\mathbf{1} = [1, \cdots, 1]^T$, and $\underline{\mathbf{x}}_i = [x_{i1}, \cdots, x_{iN}]^T \quad (i = 1, \cdots, d)$ is an N-dimensional vectors containing the ith components of all N data vectors in the dataset. Now the linear regression problem is simply to find the model parameter $\mathbf{w}$, so that the model prediction

$$\hat{\mathbf{y}} = \mathbf{X}^T\mathbf{w} \tag{5.5}$$

matches the ground truth labeling $\mathbf{y}$, so that the residual of the linear model $\mathbf{r} = \mathbf{y} - \mathbf{X}^T\mathbf{w}$ is zero, i.e., we need to solve the following linear system containing N equations of $d+1$ unknowns:

$$\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - \mathbf{X}^T\mathbf{w} = \mathbf{0} \tag{5.6}$$

If the number of data samples is equal to the number of unknown parameters, i.e., $N = d+1$, then $\mathbf{X}$ is square and invertible matrix (assuming all N samples are independent and therefore $\mathbf{X}$ has a full rank), and the equation above can be solved to get the desired weight vector:

$$\mathbf{w} = (\mathbf{X}^T)^{-1}\mathbf{y} \tag{5.7}$$

However, if there are many more data samples than the unknown parameters,

i.e., $N \gg d+1$, then $\mathbf{X}$ is not square and invertible, then $\mathbf{X}^T\mathbf{w} = \mathbf{y}$ is an over-determined linear system without a solution and the prediction $\hat{y} = \mathbf{X}^T\mathbf{w}$ will not match the ground truth labeling $\mathbf{y}$ in general. In this case we need to find an optimal solution $\mathbf{w}^*$ that minimizes the SSE (proportional to MSE):

$$\begin{aligned}\varepsilon(\mathbf{w}) &= \frac{1}{2}\sum_{n=1}^{N} r_n^2 = \frac{1}{2}||\mathbf{r}||^2 = \frac{1}{2}||\mathbf{y} - \mathbf{X}^T\mathbf{w}||^2 \\ &= \frac{1}{2}\mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}^T\mathbf{w} + \frac{1}{2}\mathbf{w}^T\mathbf{X}\mathbf{X}^T\mathbf{w} \end{aligned} \tag{5.8}$$

To find the optimal $\mathbf{w}$ that minimizes $\varepsilon(\mathbf{w})$ as the objective function, we set its gradient to zero and get the *normal equation*:

$$\mathbf{g}_\varepsilon(\mathbf{w}) = \frac{d}{d\mathbf{w}}\varepsilon(\mathbf{w}) = \mathbf{X}\mathbf{X}^T\mathbf{w} - \mathbf{X}\mathbf{y} = \mathbf{0}, \quad \text{i.e.} \quad \mathbf{X}\mathbf{X}^T\mathbf{w} = \mathbf{X}\mathbf{y} \tag{5.9}$$

Solving this equation for $\mathbf{w}$, we get the optimal weight vector:

$$\mathbf{w}^* = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y} = (\mathbf{X}^T)^{-}\mathbf{y} \tag{5.10}$$

where $(\mathbf{X}^T)^- = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}$ is the $(d+1) \times N$ pseudo-inverse (Section A.6.1) of the $N \times (d+1)$ matrix $\mathbf{X}^T$. This method for estimating the regression model parameters is called *ordinary least squares (OLS)*.

Having found the parameter $\mathbf{w}$ for the linear regression model $\hat{y} = \mathbf{x}^T\mathbf{w}$, we can predict the outputs corresponding to any unlabeled test dataset $\mathbf{X}_* = [\mathbf{x}_{1*}, \cdots, \mathbf{x}_{M*}]$:

$$\hat{\mathbf{y}}_* = \mathbf{f}_* = \mathbf{X}_*^T\mathbf{w}^* = [\mathbf{1}, \underline{\mathbf{x}}_1, \cdots, \underline{\mathbf{x}}_N] \begin{bmatrix} w_0^* \\ w_1^* \\ \vdots \\ w_d^* \end{bmatrix} = w_0^*\mathbf{1} + \sum_{i=1}^{d} w_i^*\underline{\mathbf{x}}_i \tag{5.11}$$

as a linear combination of $\mathbf{1}$ and $\{\underline{\mathbf{x}}_1, \cdots, \underline{\mathbf{x}}_d\}$.

We can show geometrically that the estimated $\hat{\mathbf{y}} = \mathbf{x}^T\mathbf{w}^*$ based on the optimal $\mathbf{w}^*$ is indeed the best fit of the training data. We first rewrite the normal equation in Eq. (5.9) as:

$$\begin{aligned}&\mathbf{X}\mathbf{X}^T\mathbf{w}^* - \mathbf{X}\mathbf{y} = \mathbf{X}\left(\mathbf{X}^T\mathbf{w}^* - \mathbf{y}\right) = \mathbf{X}\left(\hat{\mathbf{y}} - \mathbf{y}\right) \\ &= \mathbf{X}\mathbf{r} = \begin{bmatrix} 1 & \cdots & 1 \\ x_{11} & \cdots & x_{1N} \\ \vdots & \ddots & \vdots \\ x_{d1} & \cdots & x_{dN} \end{bmatrix} \mathbf{r} = \begin{bmatrix} \mathbf{1}^T \\ \underline{\mathbf{x}}_1^T \\ \vdots \\ \underline{\mathbf{x}}_d^T \end{bmatrix} \mathbf{r} = \mathbf{0} \end{aligned} \tag{5.12}$$

The last equality indicate that the residual vector $\mathbf{r}$ is perpendicular to each of the $d+1$ vectors of $\mathbf{X}^T = [\mathbf{1}, \underline{\mathbf{x}}_1, \cdots, \underline{\mathbf{x}}_d]$:

$$\mathbf{1}^T\mathbf{r} = \sum_{n=1}^{N} r_n = 0, \quad \underline{\mathbf{x}}_1^T\mathbf{r} = \cdots = \underline{\mathbf{x}}_N^T\mathbf{r} = 0 \tag{5.13}$$

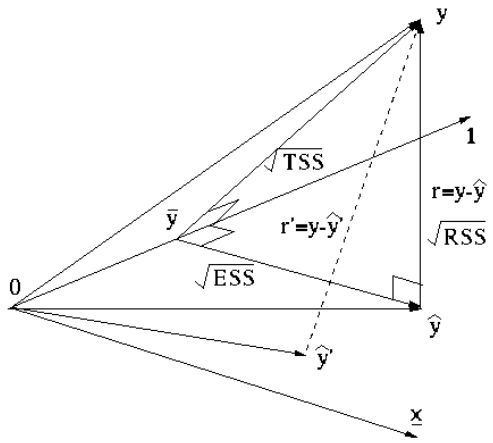


Figure 5.2 Explained, Residual and Total Sum of Squares

i.e., $\mathbf{r}$ is perpendicular to the space spanned by $\{\mathbf{1}, \underline{\mathbf{x}}_1, \cdots, \underline{\mathbf{x}}_d\}$, in which all possible estimated $\hat{\mathbf{y}} = \mathbf{x}^T\mathbf{w}$ based on any arbitrary $\mathbf{w}$ reside. We can therefore conclude that $||\mathbf{r}|| = ||\hat{\mathbf{y}} - \mathbf{x}^T\mathbf{w}^*||$ based on $\mathbf{w}^*$ is the smallest error among all possible $||\mathbf{r}'|| = ||\hat{\mathbf{y}} - \mathbf{x}^T\mathbf{w}||$ based on other $\mathbf{w} \neq \mathbf{w}^*$, i.e., $\hat{\mathbf{y}} = \mathbf{x}^T\mathbf{w}^*$ is indeed the best with minimum residual $||\mathbf{r}||^2$. This is illustrated in Fig. 5.2 for $d = 1$.

we first derive some properties of the model based on its output $\hat{\mathbf{y}} = \mathbf{X}^T\mathbf{w} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y}$:

$$\begin{aligned} ||\hat{\mathbf{y}}||^2 = \hat{\mathbf{y}}^T\hat{\mathbf{y}} &= \left(\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y}\right)^T \left(\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y}\right) \\ &= \left(\mathbf{y}^T\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\right) \left(\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y}\right) \\ &= \mathbf{y}^T\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y} = \mathbf{y}^T\hat{\mathbf{y}} \end{aligned} \tag{5.14}$$

based on which we also get the following results:

$$\mathbf{r}^T\hat{\mathbf{y}} = (\mathbf{y} - \hat{\mathbf{y}})^T\hat{\mathbf{y}} = \mathbf{y}^T\hat{\mathbf{y}} - \hat{\mathbf{y}}^T\hat{\mathbf{y}} = 0 \tag{5.15}$$

$$\begin{aligned} ||\mathbf{r}||^2 = \mathbf{r}^T\mathbf{r} &= (\mathbf{y} - \hat{\mathbf{y}})^T (\mathbf{y} - \hat{\mathbf{y}}) = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\hat{\mathbf{y}} + \hat{\mathbf{y}}^T\hat{\mathbf{y}} \\ &= \mathbf{y}^T\mathbf{y} - \mathbf{y}^T\hat{\mathbf{y}} = \mathbf{y}^T(\mathbf{y} - \hat{\mathbf{y}}) = \mathbf{y}^T\mathbf{r} \end{aligned} \tag{5.16}$$

$$||\hat{\mathbf{y}}||^2 + ||\mathbf{r}||^2 = \mathbf{y}^T\hat{\mathbf{y}} + \mathbf{y}^T\mathbf{r} = \mathbf{y}^T(\hat{\mathbf{y}} + \mathbf{r}) = \mathbf{y}^T\mathbf{y} = ||\mathbf{y}||^2 \tag{5.17}$$

We next consider how to quantitatively measure the goodness of the linear regression model $y = \mathbf{w}^T\mathbf{x}$, i.e., how well it fits the given dataset. To do so, we first define a vector $\bar{\mathbf{y}} = \bar{y}\,\mathbf{1}$ of which all components are the mean of $\{y_1, \cdots, y_N\}$:

$$\bar{y} = \frac{1}{N}\sum_{n=1}^{N} y_n = \frac{1}{N}\mathbf{y}^T\mathbf{1} \tag{5.18}$$

and then define the following three *sums of squares* of the components (squared norm) of the three vectors $\hat{\mathbf{y}} - \bar{\mathbf{y}}$, $\mathbf{y} - \hat{\mathbf{y}}$, and $\mathbf{y} - \bar{\mathbf{y}}$:

- *Explained sum of squares (ESS)* that measures the variation in the data (with respect to its mean) explained by the regression model:

$$ESS = ||\hat{\mathbf{y}} - \bar{\mathbf{y}}||^2 = \sum_{n=1}^{N} (\hat{y}_n - \bar{y})^2 \tag{5.19}$$

- *Residual sum of squares (RSS)* that represents the variation in the data not explained by the data (due to noise or discrepancy between the data and the model):

$$RSS = ||\mathbf{r}||^2 = ||\mathbf{y} - \hat{\mathbf{y}}||^2 = \sum_{n=1}^{N} (y_n - \hat{y}_n)^2 \tag{5.20}$$

- *Total sum of squares (TSS)* for the total data variation:

$$TSS = ||\mathbf{y} - \bar{\mathbf{y}}||^2 = \sum_{n=1}^{N} (y_n - \bar{y})^2 \tag{5.21}$$

We can show that the total sum of squares is the sum of the explained sum of squares and the residual sum of squares:

$$\begin{aligned} TSS &= ||\mathbf{y} - \bar{\mathbf{y}}||^2 = ||\mathbf{y} - \bar{\mathbf{y}} + \hat{\mathbf{y}} - \hat{\mathbf{y}}||^2 = ||(\hat{\mathbf{y}} - \bar{\mathbf{y}}) + (\mathbf{y} - \hat{\mathbf{y}})||^2 \\ &= ||\hat{\mathbf{y}} - \bar{\mathbf{y}}||^2 + 2(\mathbf{y} - \hat{\mathbf{y}})^T (\hat{\mathbf{y}} - \bar{\mathbf{y}}) + ||\mathbf{y} - \hat{\mathbf{y}}||^2 \\ &= ||\hat{\mathbf{y}} - \bar{\mathbf{y}}||^2 + 2\mathbf{r}^T (\hat{\mathbf{y}} - \bar{\mathbf{y}}) + ||\mathbf{r}||^2 \\ &= ESS + 2\mathbf{r}^T \hat{\mathbf{y}} - 2\mathbf{r}^T \bar{\mathbf{y}} + RSS = ESS + RSS \end{aligned} \tag{5.22}$$

The last equality is due to the fact that the two middle terms are both zero (Eqs. (5.13) and (5.15)):

$$\begin{cases} \mathbf{r}^T \bar{\mathbf{y}} = \sum_{n=1}^{N} r_n \bar{y} = \bar{y} \sum_{n=1}^{N} r_n = 0 \\ \mathbf{r}^T \hat{\mathbf{y}} = (\mathbf{y} - \hat{\mathbf{y}})^T \hat{\mathbf{y}} = \mathbf{y}^T \hat{\mathbf{y}} - \hat{\mathbf{y}}^T \hat{\mathbf{y}} = 0 \end{cases}$$

We also see that all three vectors above are perpendicular to vector $\mathbf{1}$:

$$\begin{aligned} (\hat{\mathbf{y}} - \bar{\mathbf{y}})^T \mathbf{1} &= (\hat{\mathbf{y}} - \mathbf{y} + \mathbf{y} - \bar{\mathbf{y}})^T \mathbf{1} = (\hat{\mathbf{y}} - \mathbf{y})^T \mathbf{1} + (\mathbf{y} - \bar{\mathbf{y}})^T \mathbf{1} = 0 \\ (\mathbf{y} - \hat{\mathbf{y}})^T \mathbf{1} &= \sum_{n=1}^{N} (y_n - \hat{y}_n) = \sum_{n=1}^{N} r_n = 0 \\ (\mathbf{y} - \bar{\mathbf{y}})^T \mathbf{1} &= \sum_{n=1}^{N} (y_n - \bar{y}) = N(\bar{y} - \bar{y}) = 0 \end{aligned} \tag{5.23}$$

(The second equation above is due to Eq. (5.13).) The geometry of these three vectors and the simple relationship in Eq. (5.22) is illustrated in Fig. 5.2 for $d = 1$.

We can now measure the goodness of the regression model based on the three sums of squares above. Given the total sum of squares TSS, if the model residual

RSS is small, then ESS is large, indicating most of the variation in the data can be explained by the regression model, i.e., it fits the data well and is therefore a goo model. Otherwise if RSS is large, the model is not good. Specifically, the goodness of the model can be measured by the *coefficient of determination*, denoted by R^2 (*R-squared*), as the percentage of variation in the data that is explained by the model:

$$R^2 = \frac{ESS}{TSS} = \frac{TSS - RSS}{TSS} = 1 - \frac{RSS}{TSS} \tag{5.24}$$

In general, $0 < R^2 < 1$. In the ideal case where the data is perfectly modeled by the linear regression function $f(\mathbf{x})$ with zero error $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{0}$, then $RSS = 0$, $TSS = ESS$, and $R^2 = 1$, i.e., the data is completely explained by the model. In a nonideal case with $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} \neq \mathbf{0}$, $R^2 < 1$.

How well a linear regression function models a given dataset depends on its nature such as how closely the variables in the dataset are related to each other and how noisy the dataset is, which can be measured in terms of the *correlation* between the variables. In general, the more correlated the variables, the better the dataset can be modeled, as shown below for the special case of $d = 1$. In this case how closely the two variables x and y in the 1-D dataset $\{(x_n, y_n) \ \ (n = 1, \cdots, N)\}$ are correlated can be measured by *correlation coefficient* defined as (Section B.1.3):

$$\rho = \frac{\sigma_{xy}^2}{\sqrt{\sigma_x^2 \, \sigma_y^2}} = \frac{\sigma_{xy}^2}{\sigma_x \, \sigma_y} \tag{5.25}$$

where σ_x^2 and σ_y^2 are the estimated variances of x and y:

$$\sigma_x^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})^2 = \frac{1}{N}\sum_{n=1}^{N} x_n^2 - \bar{x}^2, \quad \sigma_y^2 = \frac{1}{N}\sum_{n=1}^{N}(y_n - \bar{y})^2 = \frac{1}{N}\sum_{n=1}^{N} y_n^2 - \bar{y}^2 \tag{5.26}$$

with

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n, \quad \bar{y} = \frac{1}{N}\sum_{n=1}^{N} y_n \tag{5.27}$$

and σ_{xy}^2 is their estimated covariance:

$$\sigma_{xy}^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})(y_n - \bar{y}) = \frac{1}{N}\sum_{n=1}^{N} x_n y_n - \bar{x}\bar{y} \tag{5.28}$$

The correlation coefficient is a value between -1 and 1, it can be considered as the normalized covariance of the two variables x and y, indicating how much they are either positively or negatively correlated:

- $\rho = 1$: maximum positive correlation
- $0 < \rho < 1$: positive correlation
- $\rho = 0$: not correlated at all

- $-1 < \rho < 0$: negative correlation
- $\rho = -1$: maximum negative correlation

The model parameters of this 1-D linear regression function $y = w_0 + w_1 x$ can be obtained based on Eq. (5.10), and expressed in terms of the variances and covariance of the two variables:

$$\begin{aligned}
\mathbf{w} &= \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y} \\
&= \left(\begin{bmatrix} 1 & \cdots & 1 \\ x_1 & \cdots & x_N \end{bmatrix} \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & \cdots & 1 \\ x_1 & \cdots & x_N \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} \\
&= \begin{bmatrix} N & \sum_{n=1}^N x_n \\ \sum_{n=1}^N x_n & \sum_{n=1}^N x_n^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{n=1}^N y_n \\ \sum_{n=1}^N x_n y_n \end{bmatrix} = \begin{bmatrix} 1 & \bar{x} \\ \bar{x} & \sigma_x^2 + \bar{x}^2 \end{bmatrix}^{-1} \begin{bmatrix} \bar{y} \\ \sigma_{xy}^2 + \bar{x}\bar{y} \end{bmatrix} \\
&= \frac{1}{\sigma_x^2} \begin{bmatrix} \sigma_x^2 + \bar{x}^2 & -\bar{x} \\ -\bar{x} & 1 \end{bmatrix} \begin{bmatrix} \bar{y} \\ \sigma_{xy}^2 + \bar{x}\bar{y} \end{bmatrix} = \begin{bmatrix} \bar{y} - \sigma_{xy}^2/\sigma_x^2\, \bar{x} \\ \sigma_{xy}^2/\sigma_x^2 \end{bmatrix}
\end{aligned} \tag{5.29}$$

Based on these model parameters we get the linear regression model:

$$\hat{y} = w_0 + w_1 x = \bar{y} - \frac{\sigma_{xy}^2}{\sigma_x^2}\bar{x} + \frac{\sigma_{xy}^2}{\sigma_x^2} x = \bar{y} + \frac{\sigma_{xy}^2}{\sigma_x^2}(x - \bar{x}) \tag{5.30}$$

The coefficient of determination R^2 for the goodness of the regression model is closely related to the correlation coefficient ρ for correlation of the given data (independent of the model). Consider

$$\begin{aligned}
\frac{1}{N}RSS &= \frac{1}{N}||y - \hat{y}||^2 = \frac{1}{N}\sum_{n=1}^N (y_n - w_0 - w_1 x_n)^2 \\
&= \frac{1}{N}\sum_{n=1}^N (y_n - (\bar{y} - \bar{x}w_1) - w_1 x_n)^2 = \frac{1}{N}\sum_{n=1}^N [(y_n - \bar{y}) - w_1(x_n - \bar{x})]^2 \\
&= \frac{1}{N}\sum_{n=1}^N [(y_n - \bar{y})^2 - 2w_1(y_n - \bar{y})(x_n - \bar{x}) + w_1^2(x_n - \bar{x})^2] \\
&= \sigma_y^2 - 2w_1\sigma_{xy}^2 + w_1^2\sigma_x^2 = \sigma_y^2 - 2\frac{\sigma_{xy}^2}{\sigma_x^2}\sigma_{xy}^2 + \frac{\sigma_{xy}^4}{\sigma_x^4}\sigma_x^2 = \sigma_y^2 - \frac{\sigma_{xy}^4}{\sigma_x^2} \\
&= \sigma_y^2 \left(1 - \frac{\sigma_{xy}^4}{\sigma_x^2\sigma_y^2}\right) = \sigma_y^2(1 - \rho^2)
\end{aligned} \tag{5.31}$$

We see that if the two variables x and y are more correlated with ρ closer to 1, then the residual error $RSS = ||\mathbf{r}||^2$ is smaller and R^2 is larger, indicating the data can be better fitted by the linear regression model.

Example 5.1 In 1929, Edwin Hubble published a paper "A relationship between distance and radial velocity among extra-galactic nebulae" (Proceedings of the

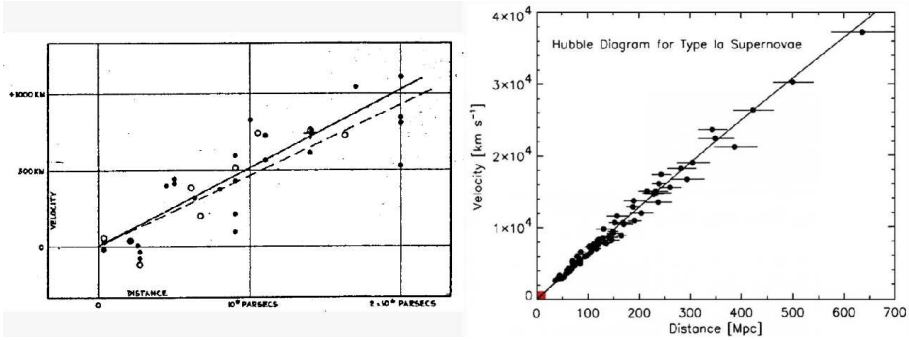


Figure 5.3 Linear Regression between recessional velocity and distance of nebulae, based on Hubble's origional data (right) and modern data (left)

National Academy of Sciences of the United states of America (PANS), Vol. 15), where the linear regression method is used to model the relationship between the distance from Earth in megaparsec ($Mpc = 3.26 \times 10^6$ light years) and the recessional velocity (in kilometer/second) of nebulae (galaxies), as shown in the left panel of Fig. 5.3. Based on this plot, Hubble estimated the slope of the linear regression, now called the *Hubble constant*

$$H_0 = \frac{\text{recessional velocity}}{\text{distance}} = \frac{v}{D}$$

to be 500 $km/sec/Mpc$. The estimated value of the Hubble constant has been continuously revised using different methods based on data with much wider range in distance collected by progressive technology. The more recent estimations of the Hubble constant are roughly in the range around 70 $km/sec/Mpc$, depending on the specific methods used (see `https://en.wikipedia.org/wiki/Hubble%27s_law`). One of such recent results is shown in the right panel of Fig. 5.3 ("Hubble's Diagram and Cosmic Expansion", R. P. Kirshner, PANS, Vol. 101 (1), Dec. 2003, Copyright (2003) National Academy of Sciences, U.S.A.).

The simple linear relationship between the recessional velocity v and distance D has led to some very profound results such as the expansion of the universe, the *Big Bang* as the beginning of the expansion, and the age of the universe since the Big Bang, which can be estimated (assuming the expansion has been linear with $H_0 = 72$ since the Big Bang, and $c = 3 \times 10^5$ km/sec) to be:

$$t_0 = \frac{1}{H_0} = \frac{D}{v} = \frac{1\ Mpc}{72\ km/sec} = \frac{3.26 \times 10^6 \times 3 \times 10^5}{72} = 13.6 \times 10^9 \text{years}$$

Example 5.2 Find a linear regression function $y = w_0 + w_1\, x$ to fit the following $N = 6$ points:

x	-3.4	-2.1	-0.8	0.3	1.7	2.5
y	-0.76	-1.04	1.75	1.82	3.17	3.15

Based on the data, we get

$$\bar{y} = 1.348, \quad \sigma_x^2 = 4.22, \quad \sigma_y^2 = 2.85, \quad \sigma_{xy}^2 = 3.27, \quad \rho = \frac{\sigma_{xy}^2}{\sigma_x \sigma_y} = 0.94$$

The augmented data array is:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -3.4 & -2.1 & -0.8 & 0.3 & 1.7 & 2.5 \end{bmatrix}$$

Solving the over-constrained equation system

$$\mathbf{y} = \begin{bmatrix} -0.76 \\ -1.04 \\ 1.75 \\ 1.82 \\ 3.17 \\ 3.15 \end{bmatrix} = \mathbf{X}^T \mathbf{w} = \begin{bmatrix} 1 & -3.4 \\ 1 & -2.1 \\ 1 & -0.8 \\ 1 & 0.3 \\ 1 & 1.7 \\ 1 & 2.5 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

we get

$$\mathbf{w} = \mathbf{X}^{-}\mathbf{y} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\,\mathbf{y}$$

$$= \begin{bmatrix} 0.13 & 0.15 & 0.16 & 0.17 & 0.19 & 0.20 \\ -0.12 & -0.07 & -0.02 & 0.02 & 0.08 & 0.11 \end{bmatrix} \begin{bmatrix} -0.76 \\ -1.04 \\ 1.75 \\ 1.82 \\ 3.17 \\ 3.15 \end{bmatrix} = \begin{bmatrix} 1.58 \\ 0.78 \end{bmatrix}$$

and the linear regression model function $y = w_0 + w_1\, x = 1.58 + 0.78\, x$, a straight line with intercept $w_0 = 1.58$, slope $w_1 = 0.78$, and normal direction $[0.78,\ -1]^T$, as shown in Fig. 5.4. The LS error is $||\mathbf{r}|| = ||\mathbf{y} - \hat{\mathbf{y}}|| = 1.38$. We further get

$$\bar{\mathbf{y}} = [1.348,\ 1.348,\ 1.348,\ 1.348,\ 1.348,\ 1.348]^T$$
$$\hat{\mathbf{y}} = \mathbf{X}^T\mathbf{w} = [-1.054,\ -0.047,\ 0.961,\ 1.813,\ 2.898,\ 3.518]^T$$

and

$$ESS = 15.21, \ \ RSS = 1.91, \ \ TSS = 17.12, \ \ R^2 = 0.89$$

We can also check that

$$\frac{1}{N} RSS = \sigma_y^2 (1 - \rho^2) = 0.318$$

Example 5.3 The two datasets given in the table below are fitted by two linear regression functions as shown in the left and right panels in Fig. 5.5.

x	-3.0	-2.5	-2.0	-1.5	-1.0	-0.5	0.0	$\cdots\cdots$
y_1	-2.67	-3.76	-0.64	-0.93	-0.01	-1.42	3.38	
y_2	2.73	-9.38	-1.44	-9.86	-2.82	-7.29	8.31	

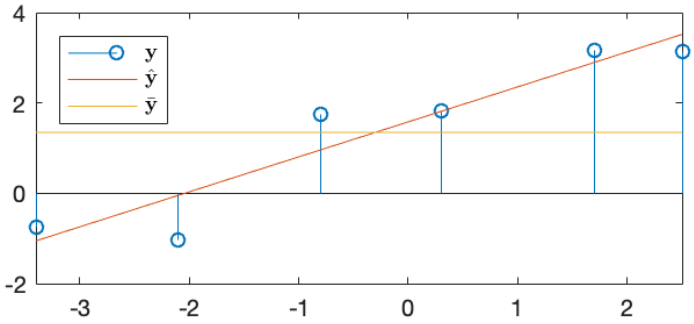


Figure 5.4 Linear Regression Example

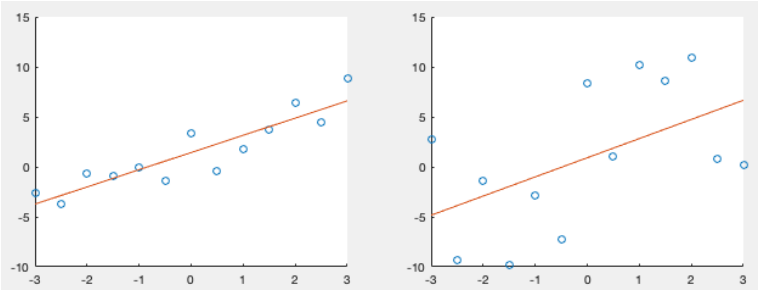


Figure 5.5 Linear Regressions with Different R^2 and ρ

	0.5	1.0	1.5	2.0	2.5	3.0
.....	−0.47	1.77	3.67	6.40	4.49	8.82
	0.98	10.18	8.58	10.86	0.77	0.20

	w_0	w_1	ESS	RSS	TSS	R^2	ρ
$dataset1$	1.43	1.72	134.71	30.18	164.89	0.817	0.904
$dataset2$	0.91	1.92	166.98	443.92	610.89	0.273	0.523

Example 5.4 Fig. 5.6 shows a set of $N = 30$ data points $\{(x_i, y_i),\ i = 1, \cdots, 30\}$ fitted by a 1-D linear regression function $y = w_0 + w_1 x$, a straight line, with $w_0 = 1.81$, $w_1 = 3.22$. The correlation coefficient is $\rho = 0.98$, and $ESS = 734.78$, $RSS = 32.98$, $TSS = 767.77$, $R^2 = 0.96$. Fig. 5.7 further shows three more examples with different correlation coefficient values $\rho = 0.982,\ 0.110,\ 0.989$, indicating the two variables are highly positively correlated, not correlated, and highly negatively correlated, respectively.

Example 5.5 Fig. 5.6 shows a set of $N = 90$ data points $\{(\mathbf{x}_i, y_i),\ i = 1, \cdots, 90\}$ fitted by a 2-D linear regression function $y = w_0 + w_1 x_1 + w_2 x_2$,

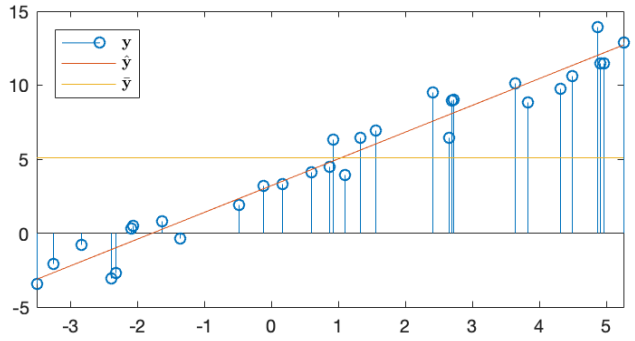


Figure 5.6 Example Showing $\mathbf{y}$, $\hat{\mathbf{y}}$ and $\bar{\mathbf{y}}$

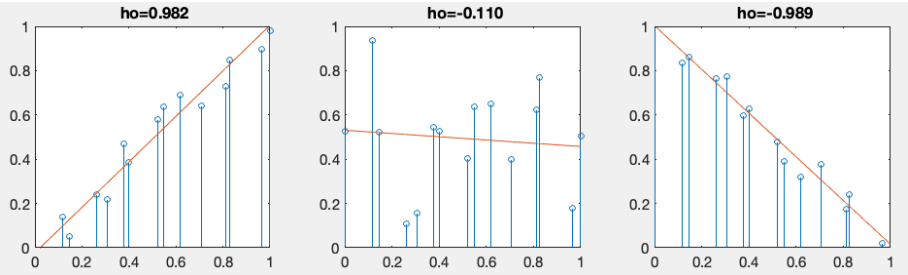


Figure 5.7 Examples of Different Correlation Coefficients

with $w_0 = 2.82,\ w_1 = 1.85,\ w_2 = 4.31$. This is a plane as shown in Fig. 5.8 from two perspective view angles. Here $ESS = 677.00, RSS = 124.48,\ TSS = 802.47$ and $R^2 = 0.84$.

Finally we note that the linear regression method can be used for binary classification, if the regression function $y = f(\mathbf{x})$, as a hyperplane in the $d+1$ dimensional space spanned by $\{x_1, \cdots, x_d, y\}$, is thresholded by a constant C, e.g., $C = 0$. The resulting equation $f(\mathbf{x}) = C$ defines a hyperplane in the d dimensional space spanned by $\{x_1, \cdots, x_d\}$, which partitions the space into two parts in such a way that all points on one side of the hyperplane satisfy $f(\mathbf{x}) > C$, while all points on the other side satisfy $f(\mathbf{x}) < C$. We see that the regression function is a binary classifier that separates all points $\mathbf{x}$ in the d dimensional space into two classes C_+ and C_-, depending on whether $f(\mathbf{x})$ is greater or smaller than C. Now each y_n corresponding to $\mathbf{x}_n$ in the given dataset $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ can be treated as a label indicating $\mathbf{x}_n$ belong to class C_+ if $y_n > C$, or C_- if $y_n < C$, and the regression problem becomes a binary classification problem. This binary classifier is illustrated in Fig. 5.1 for $C = 0$.

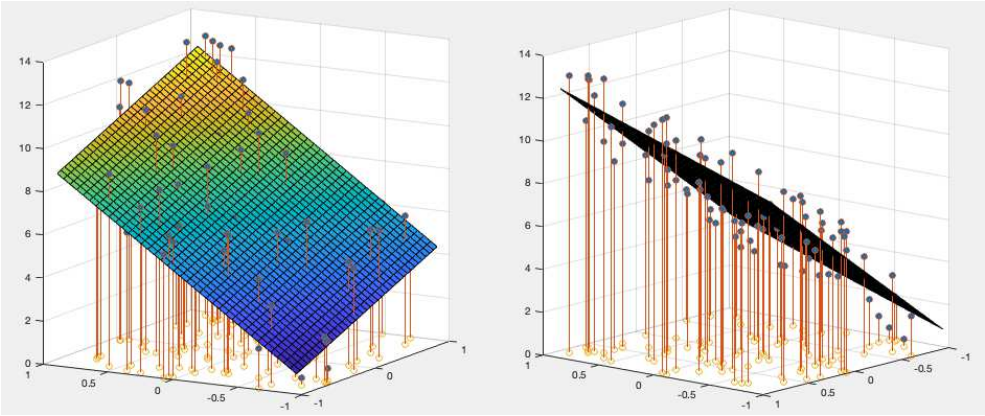


Figure 5.8 Linear Regression of a 2-D Dataset

5.2 Ridge Regression

The linear regression method considered in previous section may be prone to noise and suffer from overfitting, if the problem is ill-conditioned, when the data samples in the dataset are barely independent and therefore ill-behaved. See more detailed discussion in Section A.8.5. In such a case, matrix $\mathbf{X}\mathbf{X}^T$ still has a full rank but it is close to singularity, i.e., some of its eigenvalues may be very close to zero, and the corresponding eigenvalues of its inverse $(\mathbf{X}\mathbf{X}^T)^{-1}$ may take very large values, causing some components of the model parameter $\mathbf{w} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y}$ to also take very large values. Now any minor change in the dataset (in either $\mathbf{x}$ or y) due to noise may be amplified by $\mathbf{w}$ to cause a major change in the model output $\hat{y} = \mathbf{w}^T\mathbf{x}$. In other words, the regression model becomes highly prone to noise, i.e., it may overfit a given dataset with a large variance error thereby causing poor generalizability.

This issue of ill-conditioning can be illustrated in the 2-D case, where a linear regression function $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, a 3-D plane, is to be determined to fit a set of given data points $\{(\mathbf{x}_n, y_n),\ n = 1, \cdots, N\}$. If two variables x_1 and x_2 in the dataset are highly correlated, i.e., they are narrowly distributed in the 2-D space around a straight line (similar to the case in either the first or third panel in Fig. 5.7), then the value of $|\rho|$ is large and close to 1, and some eigenvalues of $\mathbf{X}\mathbf{X}^T$ take small values close to 0, and the 2-D regression problem becomes ill-conditioned as it degenerates into a 1-D problem. If y is linearly related to x_1 and x_2, then all data points $\{(\mathbf{x}_n, y_n),\ n = 1, \cdots, N\}$ are closely distributed around a straight line in the 3-D space which can be fitted by any 3-D plane through the line with drastically different orientations (rotated differently around the 3-D line as an axis). Consequently it becomes difficult to determine a unique 3-D plane to fit the data as it may change drastically due to minor variations in y_n or $\mathbf{x}_n$.

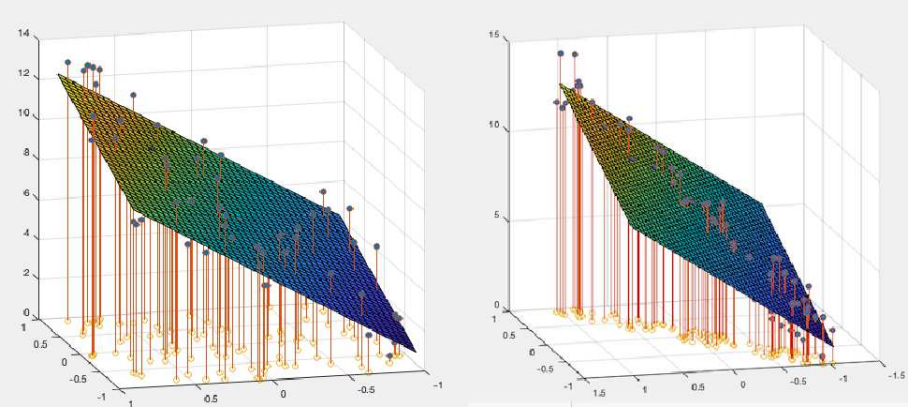


Figure 5.9 2-D Linear Regression, Well-conditioned (left) versus Ill-Conditioned (right)

Example 5.6 Fig. 5.9 compares two different datasets one well behaved while the other ill-behaved. In the first case shown in the left panel, the two independent variables x_1 and x_2 are widely distributed in the 2-D plane, i.e., they are uncorrelated with a small $\rho = -0.322$, and the matrix $\mathbf{X}\mathbf{X}^T$ has two large eigenvalues $\lambda_1 = 12.34$ and $\lambda_2 = 25.44$. This is a well-conditioned problem that results in a reliable regression plane as shown. In the second case shown in the right panel x_1 and x_2 are narrowly distributed around a straight line in the 2-D plane, i.e., they are highly correlated with a large $\rho = 0.99$, and one of the two eigenvalues $\lambda_1 = 0.09$ and $\lambda_2 = 43.24$ of matrix $\mathbf{X}\mathbf{X}^T$ is close to zero. This is a ill-conditioned problem that results in an unstable regression plane that can be drastically affected (rotating around a straight line in the 3-D space) due to noise in the data.

To address the ill-conditioned problem in linear regression, the method of *ridge regression* can be used to regularize the problem by imposing some additional constraints to prevent $\mathbf{w}$ from taking large values while minimizing the squared error SSE in Eq. (5.8). To do so, we construct an objective function $J(\mathbf{w})$ to be minimized that contains not only the SSE term $\varepsilon = ||\mathbf{r}||^2/2$, but also a regularization or penalty term $\lambda||\mathbf{w}||^2/2$ to discourage large values in $\mathbf{w}$:

$$J(\mathbf{w}) = \frac{1}{2}||\mathbf{y} - \mathbf{X}^T\mathbf{w}||^2 + \frac{\lambda}{2}||\mathbf{w}||^2 \tag{5.32}$$

so that the optimal $\mathbf{w}$ that minimizes this objective function has a small norm $||\mathbf{w}||$ as well as a small SSE. This method is called *weight decay* and the hyperparameter λ is called the *weight decay parameter*. Similar to Eq. (5.9), we set its gradient to zero:

$$\frac{d}{d\mathbf{w}}J(\mathbf{w}) = \mathbf{X}\mathbf{X}^T\mathbf{w} - \mathbf{X}\mathbf{y} + \lambda\mathbf{w} = (\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})\mathbf{w} - \mathbf{X}\mathbf{y} = \mathbf{0} \tag{5.33}$$

and solve the resulting equation to get

$$\mathbf{w}^* = (\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})^{-1}\mathbf{X}\mathbf{y} \tag{5.34}$$

By adjusting the hyperparameter λ, we can make a proper tradeoff between stability of the result and its accuracy in terms of the goodness of the model measured by R^2):

- If λ is small, the result is more accurate but less stable as it is more prone to noise, i.e., it tends to overfit the data with a large bias error but small variance.
- If λ is large, the result is less affected by noise and therefore more stable, but it is less accurate, i.e., it tends to underfit the data with a small bias error but a large variance error.

In general, to prevent overfitting, regularization by means of including in the objective function a constraint term to impose some additional constraints over the model parameters is commonly used in many supervised learning algorithms for both regression and classification, including the classification methods of support vector machine in Section 14 and the back propagation neural network in Section 18.2.

Example 5.7 The method of ridge regression is applied to the same ill-conditioned dataset in the right panel in Fig. 5.9, where the two independent variables x_1 and x_2 are highly correlated with $\rho = 0.99$. As the result, matrix $\mathbf{X}\mathbf{X}^T$ is close to singularity with one of its two eigenvalues $\lambda_1 = 43.24$ and $\lambda_2 = 0.09$ close to zero, i.e., the matrix is close to singularity and the problem is ill-conditioned.

When the regular linear regression ($\lambda = 0$) is applied, it produces the results shown in the top two panels in Fig. 5.10 for the dataset before and after the dependent variable y is slightly perturbed. We see that the orientations of the two resulting regression planes differ drastically, due to the perturbance in y. But when the ridge regression method (with $\lambda = 1$) is applied, it produces more stable results as shown in the bottom two panels of the figure. We see that the same perturbation in the dataset caused only a minor change in the orientation of the regression plane.

The cost of such more stable results is their accuracy, as the model goodness values R^2 for the dataset before and after the perturbance are reduced from 0.894 and 0.915 for the regular regression to 0.857 and 0.876 respectively for the ridge regression.

This example shows that the stability of the solution of an ill-conditioned problem can be improved with the cost of lower accuracy (smaller R^2), and tradeoff between stability (low bias error) and accuracy (low variance error) can be made by adjusting parameter λ.

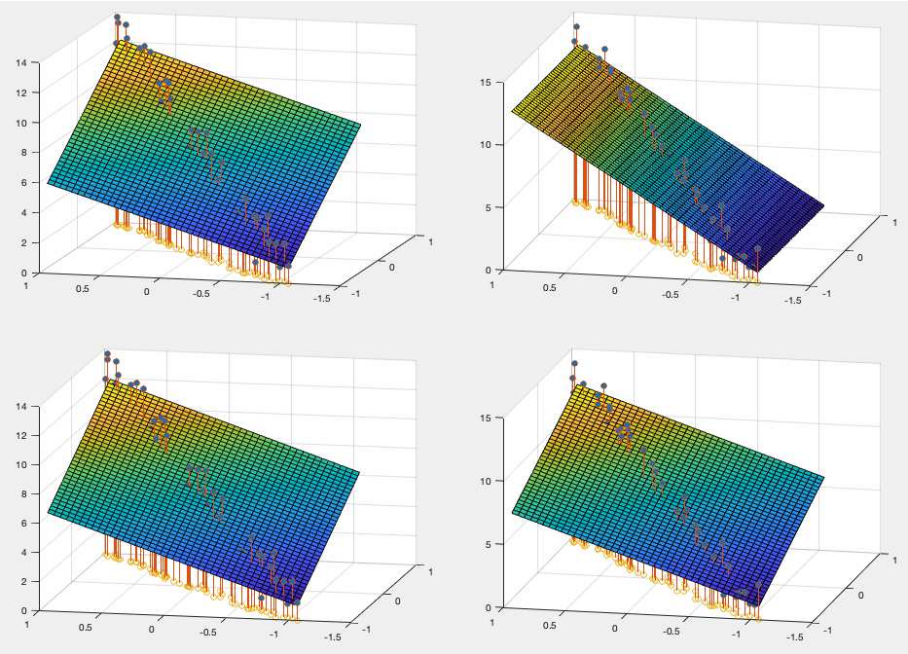


Figure 5.10 Unstable Results Generated by Regression applied to Ill-Conditioned Dataset (top) and Stable Results Generated by Ridge Regression (bottom)

5.3 Regression Based on Basis Functions

The method of linear regression considered previously can be generalized to model nonlinear relationships between the dependent variable y and the independent variables in $\mathbf{x} = [x_1, \cdots, x_d]^T$ by a regression function as a linear combination of K nonlinear basis functions of $\mathbf{x}$:

$$\hat{y} = f(\mathbf{x}) = \sum_{k=1}^{K} w_k \, \varphi_k(\mathbf{x}, \boldsymbol{\theta}) = \boldsymbol{\varphi}(\mathbf{x})^T \mathbf{w} \tag{5.35}$$

where $\varphi_k(\mathbf{x}, \boldsymbol{\theta}),\ (k = 1, \cdots, K)$ is one of the basis function parameterized by $\boldsymbol{\theta}$ that span the function space in which y resides, and $\boldsymbol{\varphi}(\mathbf{x}) = [\varphi_1(\mathbf{x}), \cdots, \varphi_K(\mathbf{x})]^T$ is a K-D vector. Some of the commonly used basis functions are listed below and plotted in Fig. 5.11 for $K = 7$.

- Power function (top-left):

$$\varphi_k(x) = x^{k-1} \tag{5.36}$$

  This is the $(k-1)$th power of x, and the regression function $f(x)$ is $(K-1)$th order polynomial of x.
- Sinusoidal function (top-right):

$$\varphi_k(x, \omega) = \sin((k-1)\omega x), \quad \text{or} \quad \varphi_k(x, \omega) = \cos((k-1)\omega x) \tag{5.37}$$

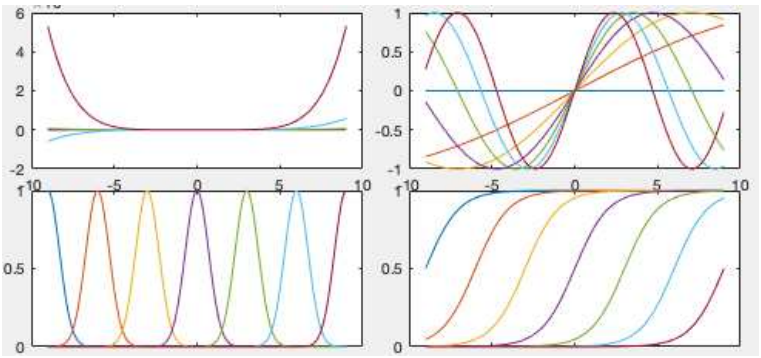


Figure 5.11 Basis Functions for Regression

This is the $(k-1)$th harnomic of a sinusoidal function of x, and the regression function $f(x)$ is the first K terms of the sine or cosine series expansion of the given function.

- Gaussian function (bottom-left):

$$\varphi_k(\mathbf{x}, \mathbf{c}_k, a) = \exp(a||\mathbf{x} - \mathbf{c}_k||^2) \tag{5.38}$$

This is a member of a family of functions called *radial basis functions (RBFs)*, each centered at $\mathbf{c}_k$.

- Sigmoid function (bottom-right):

$$\varphi_k(\mathbf{x}, \mathbf{c}_k, a) = \frac{1}{1 + e^{-a||\mathbf{x}-\mathbf{c}_k||}} \tag{5.39}$$

In practice, choosing a proper type of basis functions may be a trial-and-error process depending on the specific dataset.

The original d-dimensional data space spanned by $x_1, \cdots, x_d$ is now converted to a K-dimensional space spanned by the K basis functions $\varphi_1(\mathbf{x}), \cdots, \varphi_K(\mathbf{x})$. Applying such a regression model to each data point $\mathbf{x}_n$, we get

$$\hat{y}_n = \sum_{k=1}^{K} w_k \, \varphi_k(\mathbf{x}_n) = \boldsymbol{\varphi}(\mathbf{x}_n)^T \mathbf{w} \qquad (n = 1, \cdots, N) \tag{5.40}$$

These N equations can be written in matrix form:

$$\hat{\mathbf{y}} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} \varphi^T(\mathbf{x}_1) \\ \vdots \\ \varphi_K^T(\mathbf{x}_N) \end{bmatrix} \mathbf{w} = \boldsymbol{\Phi}^T \mathbf{w} \tag{5.41}$$

where $\boldsymbol{\Phi}$ is a $K \times N$ matrix:

$$\boldsymbol{\Phi} = [\varphi(\mathbf{x}_1), \cdots, \varphi(\mathbf{x}_N)] = \begin{bmatrix} \varphi_1(\mathbf{x}_1) & \cdots & \varphi_1(\mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ \varphi_K(\mathbf{x}_1) & \cdots & \varphi_K(\mathbf{x}_N) \end{bmatrix} \tag{5.42}$$

Typically there are many more sample data points than the basis functions, i.e., $N \gg K$.

If, specially, $\varphi(u) = u$ is an identity function, then Eq. (5.35) becomes the same as Eq. (5.2) for linear regression. Also, as the nonlinear model $\hat{\mathbf{y}} = \mathbf{\Phi}^T\mathbf{w}$ in Eq. (5.41) is in the same form as the linear model $\hat{\mathbf{y}} = \mathbf{X}^T\mathbf{w}$ in Eq. (5.2), it can be solved the same way as in Eq. (5.10), by simply replacing $\mathbf{X}$ by $\mathbf{\Phi}$:

$$\mathbf{w}^* = (\mathbf{\Phi}\mathbf{\Phi}^T)^{-1}\mathbf{\Phi}\mathbf{y} = (\mathbf{\Phi}^T)^-\mathbf{y} \tag{5.43}$$

where $(\mathbf{\Phi}^T)^- = (\mathbf{\Phi}\mathbf{\Phi}^T)^{-1}\mathbf{\Phi}$ is the $K \times N$ pseudo-inverse of the $N \times K$ matrix $\mathbf{\Phi}^T$. Now we get the output of the regression model

$$\hat{\mathbf{y}} = \mathbf{\Phi}^T\mathbf{w} = \mathbf{\Phi}^T\mathbf{\Phi}^-\mathbf{y} = \mathbf{\Phi}^T(\mathbf{\Phi}\mathbf{\Phi}^T)^{-1}\mathbf{\Phi}\mathbf{y} \tag{5.44}$$

The Matlab code segment below shows the essential part of the algorithm, where $x(n)$ is the nth data sample $\mathbf{x}_n$ out of the total of N, $y(x)$ is the vector $\mathbf{y} = [y_1, \cdots, y_N]^T$ with $y_n = f(\mathbf{x}_n)$, and $c(k)$ contains the parameters for the kth basis function $\phi_k(\mathbf{x}) = \phi(\mathbf{x}, c(k))$ represented by `phi` in the code.

```
for n=1:N
    for k=1:K
        Phi(k,n)=phi(x(n),c(k));
    end
end
w=pinv(Phi')*y(x);      % weight vector by LS method
yhat=Phi'*w;            % reconstructed function
```

Example 5.8 The plots in Figs. 5.12 through 5.15 show the approximations of a given function based on a set of N samples using linear regression based on the four different types of basis functions. We see that as the number of basis functions K increases, the approximation becomes more accurate.

Example 5.9 The linear regression method based on basis functions is shown in Fig. 5.16, where a 2-D function $f(\mathbf{x}) = f(x_1, x_2)$ (left) is to be approximated based on the function values in $\{y_1, \cdots, y_N\}$ at a set N sample points $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ taken from a grid in the 2-D space (right). The regression is based on K 2-D Gaussian basis functions centered at different locations in the 2-D space shown in Fig. 5.17 for $K = 9$.

The regression results based on different numbers of basis functions are shown in Fig. 5.18. Again, we see that the approximation becomes more accurate as the value of K increases.

In addition to the different types of basis functions considered above, we fur-

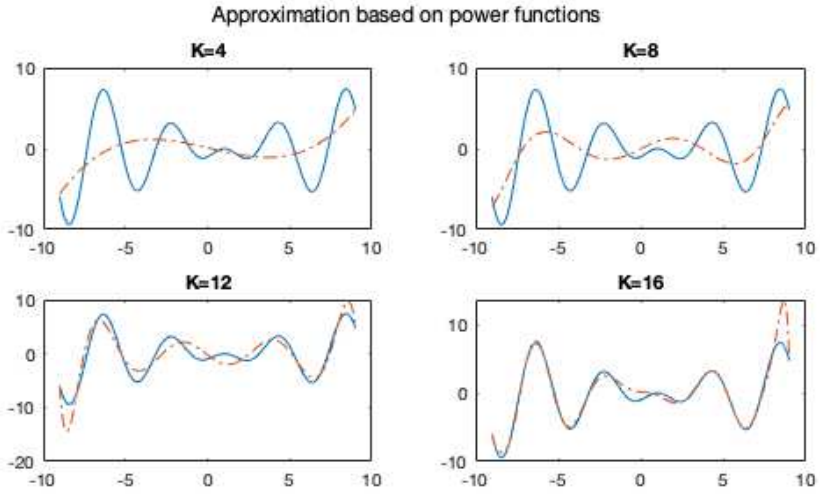


Figure 5.12 Regression based on Power Basis

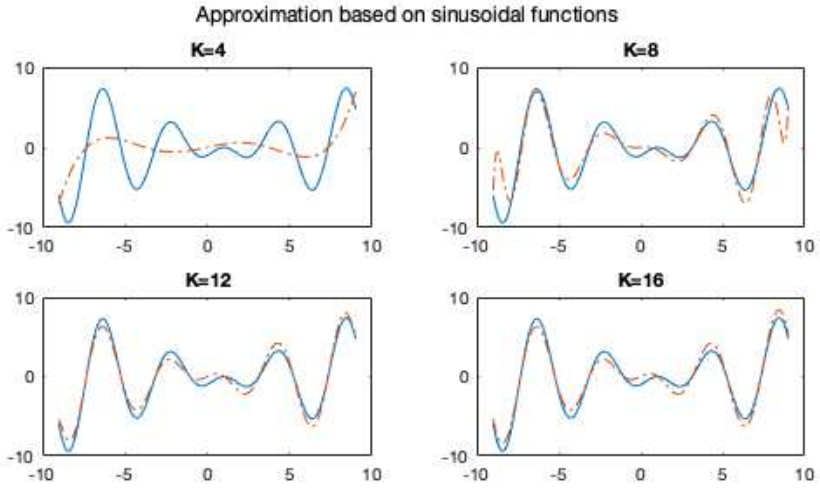


Figure 5.13 Regression based on Sinusoidal Basis

ther consider the following basis functions:

$$\varphi_k(\mathbf{x}, \mathbf{w}_k, b_k) = g\left(\mathbf{w}_k^T \mathbf{x} + b_k\right) = g\left(\sum_{i=1}^{d} w_{ki} x_i + b_k\right) = g\left(\sum_{i=0}^{d} w_{ki} x_i\right) \quad (5.45)$$

where $g(x)$, called *activation function*, can take some different forms such as a

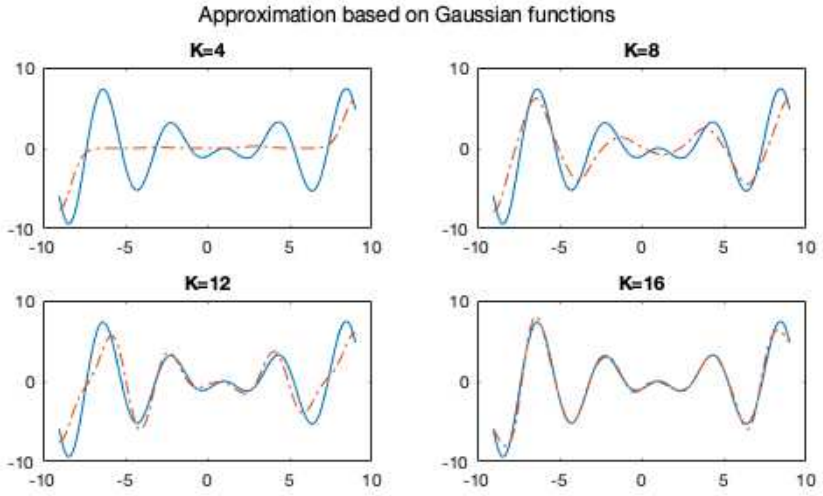


Figure 5.14 Regression based on Gaussian Basis

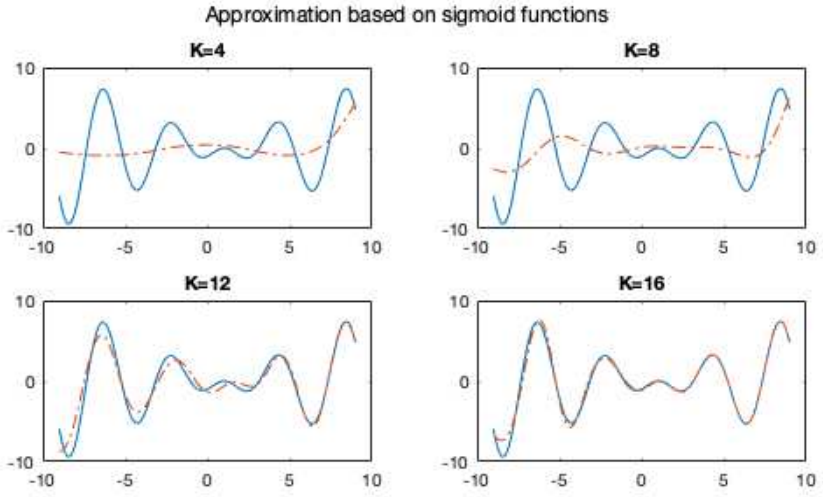


Figure 5.15 Regression based on Sigmoid Basis

sigmoid or rectified linear function (rectified linear unit or ReLU),

$$g(x) = \begin{cases} 1/(1+e^{-x}) & \text{sigmoid function} \\ \max(0,x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} & \text{ReLU} \end{cases} \tag{5.46}$$

Now the regression function in Eq. (5.35) can be written as below for multiple

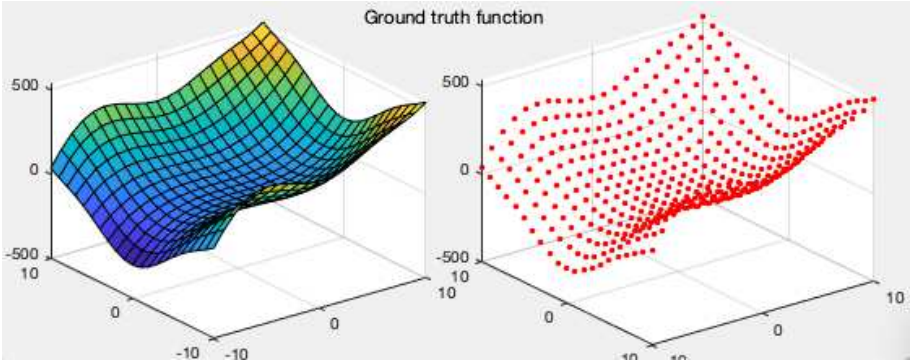


Figure 5.16 A Function and Its Samples in 2-D

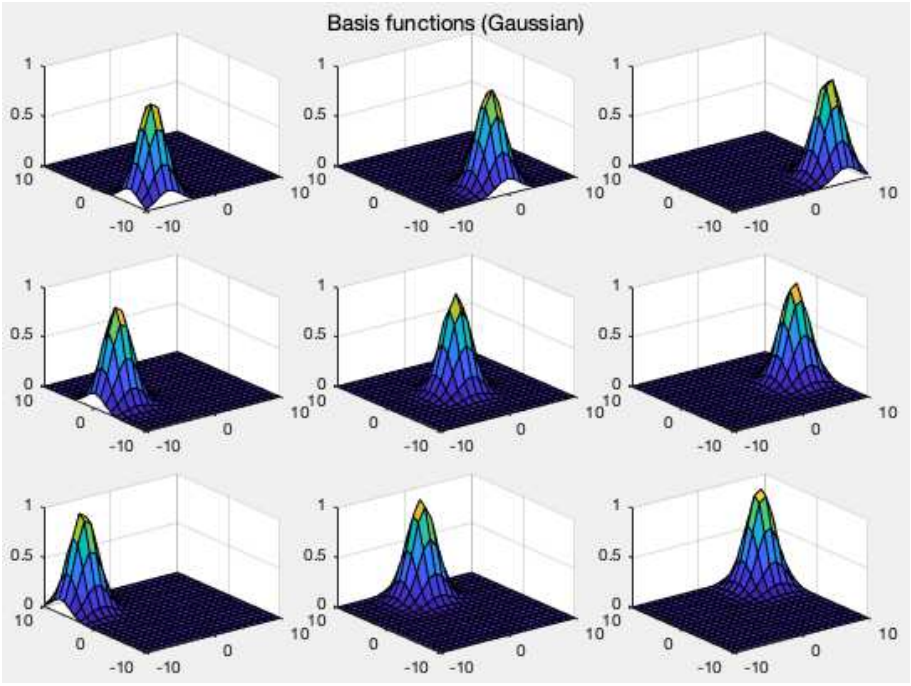


Figure 5.17 Gaussian Basis in 2-D

outputs :

$$\hat{y}_j = \sum_{k=1}^{K} w_{jk}^{o}\, \varphi_k(\mathbf{x}, \boldsymbol{\theta}_k) = \sum_{k=1}^{K} w_{jk}^{o}\, g\left(\sum_{i=0}^{d} w_{ki}^{h} x_i\right) \qquad (j = 1, \cdots, m) \qquad (5.47)$$

which can be interpreted as the output of a three-layer *neural network* shown in Fig. 5.19 containing an input layer of d nodes, a middle or hidden layer of K nodes with weights w_{ki}^{h}, $(i = 0, \cdots, d,\ k = 1, \cdots, K)$, and an output layer of

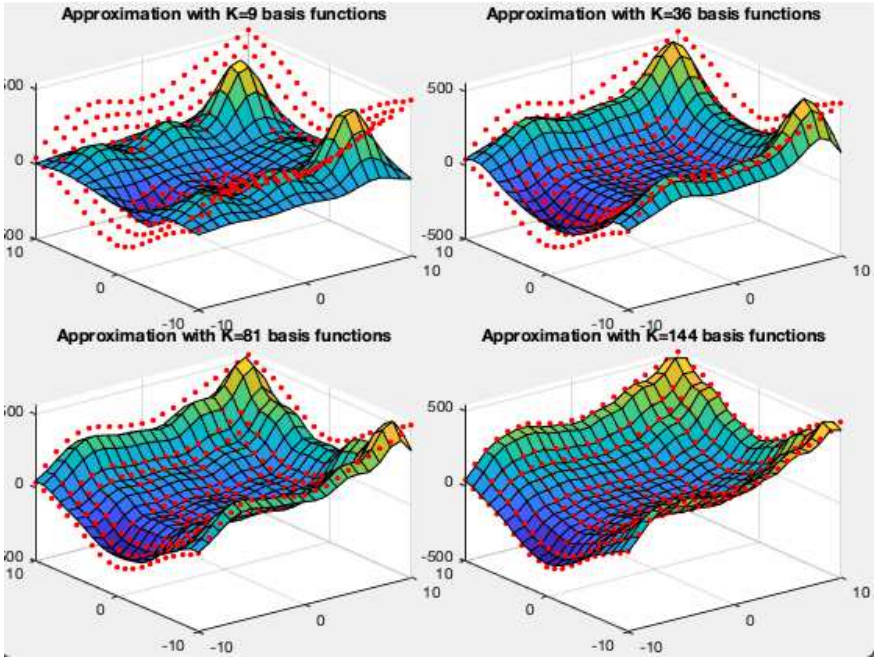


Figure 5.18 Regression Results based on Different Number of Basis Functions

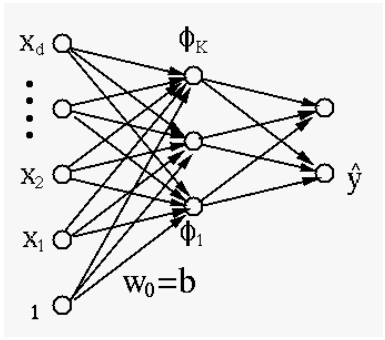


Figure 5.19 A Network Model for Regression

m nodes with weights w_{jk}^o, $(k = 1, \cdots, K,\ j = 1, \cdots, m)$. These nodes are also called *neurons*, as they mimick the biological neurons in the brain.

As shown in Fig. 5.19, the nodes of the input layer take the d components of a data point $\mathbf{x}$ from the dataset one at a time, while each of the K nodes of the hidden layer calculates their linear combination $\mathbf{w}_k^T\mathbf{x}$, and, together with a bias term b_k, generates the basis function $\varphi_k(\mathbf{x}) = g\left(\mathbf{w}_k^T\mathbf{x} + b_k\right)$ as its output. Finally, each node of the output layer takes these K basis function from the K hidden layer and generates as its output the regression function $\hat{y} = f(\mathbf{x}) = \sum_{k=1}^{K} w_k\varphi_k(\mathbf{x})$ as the model of the dataset.

This network depends on two sets of weights, those represented by w_{ki}^h for the hidden nodes, and those represented by w_{jk}^o (found in Eq. (5.43)) for the output nodes. When there are multiple nodes in the output layer, they can each approximate a different function. This three-layer network is of particular interest in the general method of artificial neural networks to be considered in Chapter 17.

Example 5.10 The method of regression based on basis functions is used to approximate the same 2-D function $f(\mathbf{x}) = f(x_1, x_2)$ in the previous example, as a linear combination of K basis functions $\varphi_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + b_k$ with each parameterized by $\mathbf{w}_k$ and b_k (randomly generated).

The Matlab code segment below shows the essential part of the program for this example, where f is a arbitrary function to be approximated.

```
a=@(x) x.*(x>0);      % ReLU activation function, or
a=@(x) 1/(1+exp(-x)); % Sigmoid activation function
g=@(x,w,b)a(w'*x+b);  % the basis function

[X,Y]=meshgrid(xmin:1:xmax, ymin:1:ymax); % define 2-D points
nx=size(X,2);         % number of samples in first dimension
ny=size(Y,1);         % number of samples in second dimension
N=nx*ny;              % total number of data samples
Phi=zeros(N,K);      % K basis functions evaluated at N sample points
W=1-2*rand(2,K);      % random initialization of weights
b=1-2*rand(1,K);      % and biases for K basis functions
n=0;
for i=1:nx
    x(1)=X(1,i);      % first component
    for j=1:ny
        x(2)=Y(j,1);  % second component
        n=n+1;        % of the nth sample
        Y(i,j)=func(x); % function evalued at the sample
        for k=1:K       % basis functions evaluated at the sample
            Phi(k,n)=g(x,W(:,k),b(k));
        end
    end
end
y=reshape(Y,N,1);     % convert function values to vector
w=pinv(Phi')*y;       % find weights by LS method
yhat=Phi'*w;          % reconstructed function
Yhat=reshape(yhat,m,n); % convert vector y to 2-D to compare w/ Y
```

Figs. 5.20 through 5.23 show the basis functions based on both the sigmoid and ReLU activation functions, and the corresponding regression results. We

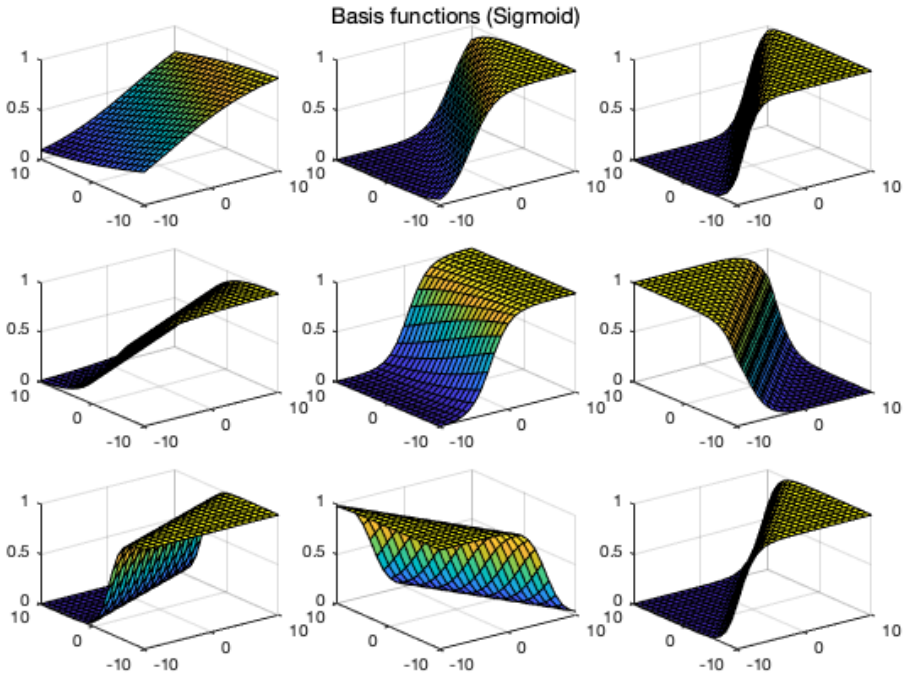


Figure 5.20 Sigmoid Basis for 2-D Regression

note that as more basis functions are used, the more accurately the function is approximated.

5.4 Bayesian Regression

The regression algorithms previously considered frequentist methods that treat all data and parameters in a regression model as deterministic variables. However, regression can also be viewed in light of Bayesian inference, where all data and parameters in the regression model are treated as random variables. In this section, we consider two such methods, the maximum likelihood estimation (MLE) and the maximum a posterior (MAP) estimation. While the MLE method turns out to be equivalent to the least squares method, the MAP method will generate results including not only the estimated model parameter $\mathbf{w}$ but also the variance of the estimate as a confidence measurement. These methods can also be used for nonlinear regression based on basis functions.

Here we assume the model parameter $\mathbf{w}$ in the linear regression function $\hat{y} = f(\mathbf{x}, \mathbf{w}) = \mathbf{x}^T\mathbf{w}$ is a random vector. Consequently, the regression function $f(\mathbf{x}, \mathbf{w})$, as a function of $\mathbf{w}$, is also random, so is the residual $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - \mathbf{X}^T\mathbf{w}$

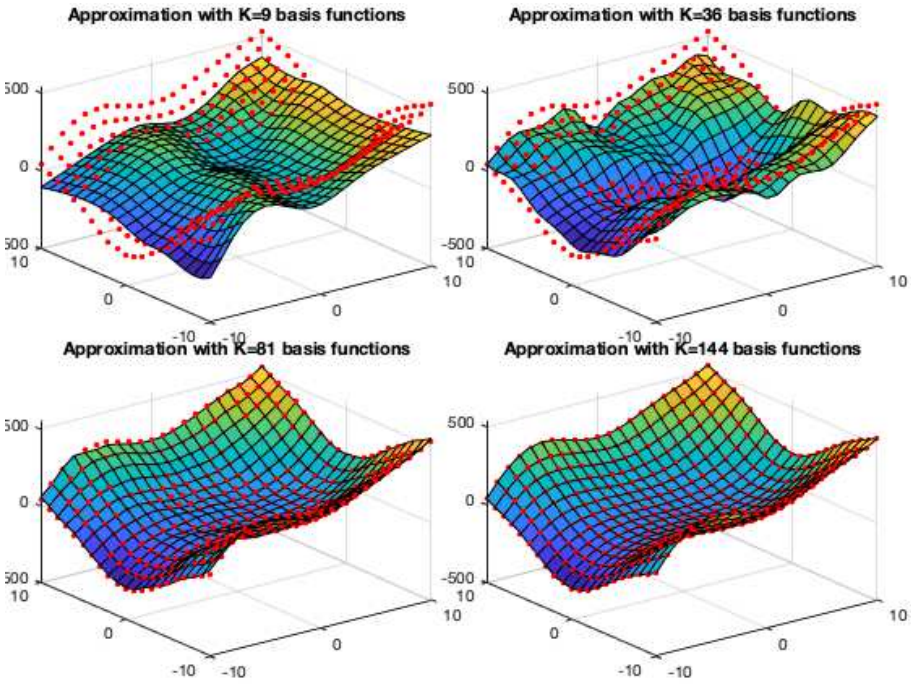


Figure 5.21 Regression based on Sigmoid Basis

based on the training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$. Now we have

$$\mathbf{y} = \hat{\mathbf{y}} + \mathbf{r} = \mathbf{f}(\mathbf{X}) + \mathbf{r} = \mathbf{X}^T\mathbf{w} + \mathbf{r} \tag{5.48}$$

We assume $\mathbf{w}$ has a zero-mean normal prior pdf:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w},\, \mathbf{0},\, \boldsymbol{\Sigma}_w) \propto \exp\left(-\frac{1}{2}\mathbf{w}^T\boldsymbol{\Sigma}_w^{-1}\mathbf{w}\right), \tag{5.49}$$

based on our desire that the weights in $\mathbf{w}$ should take values around zero, instead of large values on either positive or negative, to avoid potential overfitting. Note that here the normalizing factor independent of $\mathbf{w}$ is dropped for simplicity. We further assume that all components of the residual $\mathbf{r}$ are independent of each other and they have the same variance, i.e., $\boldsymbol{\Sigma}_r = \sigma_r^2\mathbf{I}$, and $\mathbf{r}$ has a zero-mean normal distribution:

$$p(\mathbf{r}) = \mathcal{N}(\mathbf{r},\, \mathbf{0},\, \sigma_r^2\mathbf{I}) \propto \exp\left(-\frac{1}{2\sigma_r^2}\mathbf{r}^T\mathbf{r}\right) = \exp\left(-\frac{1}{2\sigma_r^2}||\mathbf{r}||^2\right) \tag{5.50}$$

As a linear transformation of $\mathbf{w}$, the regression function $\hat{\mathbf{y}} = \mathbf{X}^T\mathbf{w}$ also has a zero-mean Gaussian distribution, and so is $\mathbf{y} = \mathbf{X}^T\mathbf{w} + \mathbf{r}$ as the sum of two Gaussians.

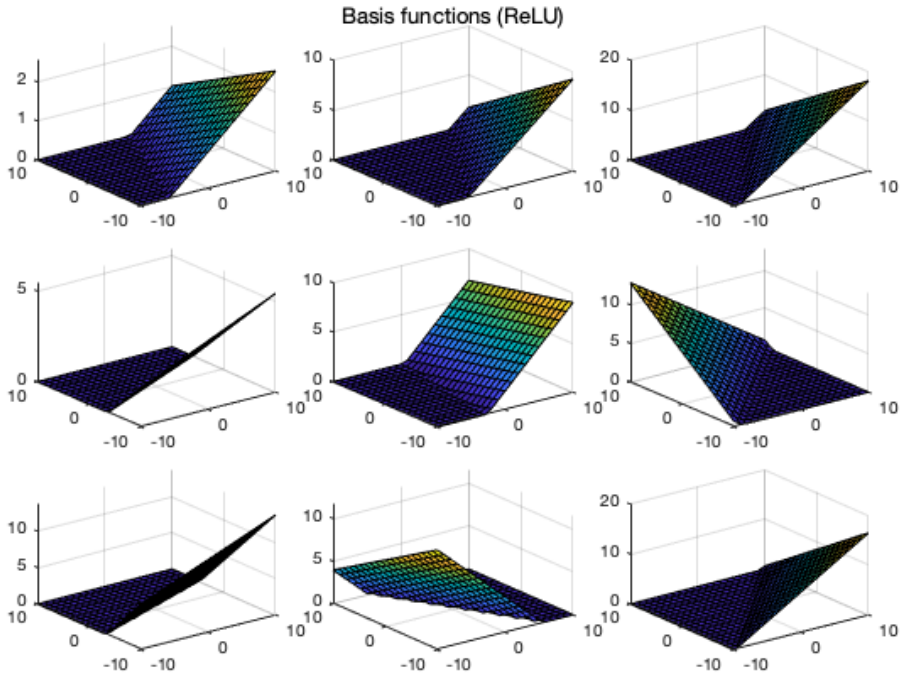


Figure 5.22 ReLU Basis for 2-D Regression

Given the observed data $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$, we get its *likelihood*:

$$\begin{aligned} L(\mathbf{w}|\mathcal{D}) \propto p(\mathcal{D}|\mathbf{w}) = p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = p(\mathbf{X}^T\mathbf{w} + \mathbf{r}) = \mathcal{N}(\mathbf{y}, \mathbf{X}^T\mathbf{w}, \sigma_r^2\mathbf{I}) \\ \propto \exp\left(-\frac{1}{2\sigma_r^2}(\mathbf{y} - \mathbf{X}^T\mathbf{w})^T(\mathbf{y} - \mathbf{X}^T\mathbf{w})\right) = \exp\left(-\frac{1}{2\sigma_r^2}||\mathbf{r}||^2\right) \end{aligned} \tag{5.51}$$

We see that the likelihood is a normal pdf of $\mathbf{y}$ (the same normal pdf but shifted by $\mathbf{X}^T\mathbf{w}$), which is the same as that of $\mathbf{r}$ given above. Now we can used the *maximum likelihood estimation (MLE)* method to find the optimal parameter $\mathbf{w}^*$ by which the likelihood $L(\mathbf{w}|\mathcal{D})$ is maximized, or equivalently the following negative log likelihood is minimized:

$$-l(\mathbf{w}|\mathbf{X}, \mathbf{y}) = -\log L(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{||\mathbf{r}||^2}{2\sigma_r^2} \propto ||\mathbf{r}||^2 \tag{5.52}$$

This is the same as the squared error $\varepsilon(\mathbf{w})$ defined in Eq. (5.8) to be minimized by the LS method previously considered. We therefore see that the maximum likelihood method and the least squares method are equivalent to each other.

We can further find the posterior $p(\mathbf{w}|\mathcal{D})$ of the model parameter $\mathbf{w}$ given the observed data $\mathcal{D}$, as the product of prior $p(\mathbf{w})$ and likelihood $L(\mathbf{w}|\mathcal{D}) \propto$

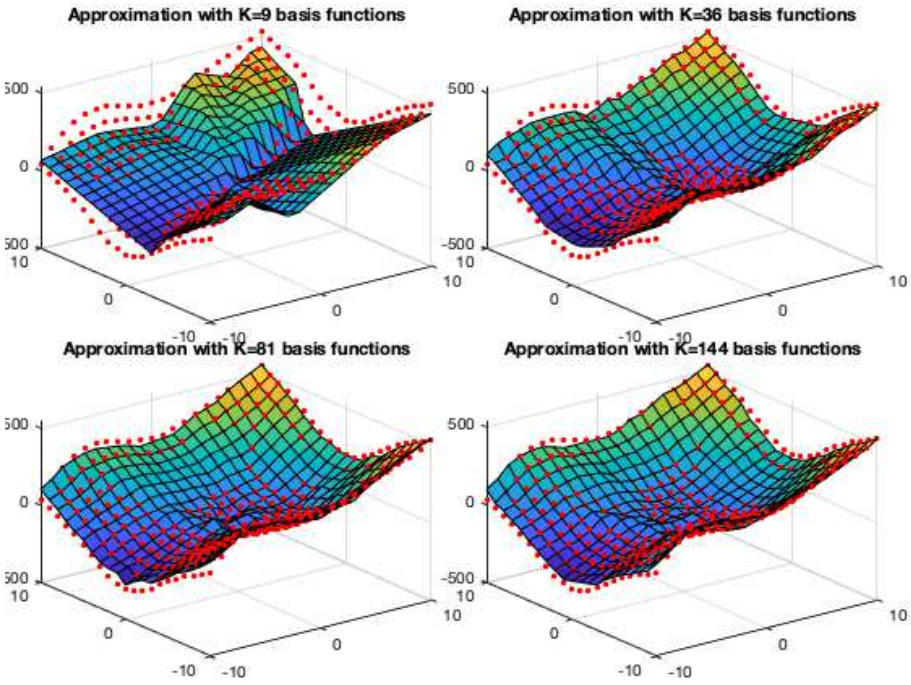


Figure 5.23 Regression based on ReLU Basis

$p(\mathbf{y}|\mathbf{X}, \mathbf{w})$:

$$
\begin{aligned}
p(\mathbf{w}|\mathcal{D}) = p(\mathbf{w}|\mathbf{X}, \mathbf{y}) &= \frac{p(\mathcal{D}|\mathbf{w})\, p(\mathbf{w})}{p(\mathcal{D})} = \frac{p(\mathbf{y}|\mathbf{X}, \mathbf{w})\; p(\mathbf{w})}{p(\mathbf{y}|\mathbf{X})} \\
&\propto p(\mathbf{y}|\mathbf{X}, \mathbf{w})\; p(\mathbf{w}) = \mathcal{N}(\mathbf{y}, \mathbf{X}^T\mathbf{w}, \sigma_r^2\mathbf{I})\; \mathcal{N}(\mathbf{w}, \mathbf{0}, \boldsymbol{\Sigma}_w) \\
&\propto \exp\left[-\frac{1}{2}\left(\frac{1}{\sigma_r^2}(\mathbf{y} - \mathbf{X}^T\mathbf{w})^T(\mathbf{y} - \mathbf{X}^T\mathbf{w}) + \mathbf{w}^T\boldsymbol{\Sigma}_w^{-1}\mathbf{w}\right)\right] \\
&\propto \exp\left[-\frac{1}{2}\left(\frac{1}{\sigma_r^2}\mathbf{y}^T\mathbf{y} - \frac{2}{\sigma_r^2}(\mathbf{X}\mathbf{y})^T\mathbf{w} + \frac{1}{\sigma_r^2}\mathbf{w}^T\mathbf{X}\mathbf{X}^T\mathbf{w} + \mathbf{w}^T\boldsymbol{\Sigma}_w^{-1}\mathbf{w}\right)\right] \\
&\propto \exp\left[-\frac{1}{2}\left(\mathbf{w}^T\left(\frac{1}{\sigma_r^2}\mathbf{X}\mathbf{X}^T + \boldsymbol{\Sigma}_w^{-1}\right)\mathbf{w} - \frac{2}{\sigma_r^2}(\mathbf{X}\mathbf{y})^T\mathbf{w}\right)\right] \qquad (5.53)
\end{aligned}
$$

As before, here all constant scaling factors (irrelevant to $\mathbf{w}$) are dropped for simplicity. As both the prior and the likelihood are Gaussian, this this posterior, when normalized, is also Gaussian in terms of the mean $\mathbf{m}_{w/d}$ and covariance $\boldsymbol{\Sigma}_{w/d}$:

$$
\begin{aligned}
p(\mathbf{w}|\mathcal{D}) = \mathcal{N}(\mathbf{w}, \mathbf{m}_{w/d}, \boldsymbol{\Sigma}_{w/d}) &\propto \exp\left[-\frac{1}{2}\left((\mathbf{w} - \mathbf{m}_{w/d})^T\boldsymbol{\Sigma}_{w/d}^{-1}(\mathbf{w} - \mathbf{m}_{w/d})\right)\right] \\
&\propto \exp\left[-\frac{1}{2}\left(\mathbf{w}^T\boldsymbol{\Sigma}_{w/d}^{-1}\mathbf{w} - 2\mathbf{m}_{w/d}^T\boldsymbol{\Sigma}_{w/d}^{-1}\mathbf{w}\right)\right] \qquad (5.54)
\end{aligned}
$$

with the following covariance and mean:

$$\begin{aligned}\boldsymbol{\Sigma}_{w/d} &= \left(\frac{1}{\sigma_r^2}\mathbf{X}\mathbf{X}^T + \boldsymbol{\Sigma}_w^{-1}\right)^{-1} \\ \mathbf{m}_{w/d} &= \frac{1}{\sigma_r^2}\boldsymbol{\Sigma}_{w/d}\mathbf{X}\mathbf{y} = \left(\mathbf{X}\mathbf{X}^T + \sigma_r^2\boldsymbol{\Sigma}_w^{-1}\right)^{-1}\mathbf{X}\mathbf{y}\end{aligned} \tag{5.55}$$

Now we can find the optimal model parameter $\mathbf{w}^* = \mathbf{m}_{w/d}$ equal to the mean of the Gaussian posterior at which it is maximized. This solution is called the *maximum a posteriori (MAP)* estimate of the model parameter $\mathbf{w}$.

We can now predict the function value of any unlabeled test data point $\mathbf{x}_*$ based on the conditional probability distribution $p(f_*|\mathbf{x}_*, \mathcal{D})$ of the regression model $\hat{y}_* = f_* = f(\mathbf{x}_*) = \mathbf{x}_*^T\mathbf{w}$, which, as a linear transformation of the random parameters in $\mathbf{w}$, is also Gaussian (Section B.1.5).

$$p(f_*|\mathbf{x}_*, \mathcal{D}) = p(\mathbf{x}_*^T\mathbf{w}|\mathbf{x}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(f_*, \mathbf{x}_*^T\mathbf{m}_{w/d},\ \mathbf{x}_*^T\boldsymbol{\Sigma}_{w/d}\mathbf{x}_*) \tag{5.56}$$

with the following mean and variance:

$$\begin{aligned}\mathrm{E}[f_*|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] &= \mathbf{x}_*^T\mathbf{m}_{w/d} = \mathbf{x}_*^T\left(\mathbf{X}\mathbf{X}^T + \sigma_r^2\boldsymbol{\Sigma}_w^{-1}\right)^{-1}\mathbf{X}\mathbf{y} \\ \mathrm{Var}[f_*|\mathbf{x}_*, \mathbf{X}, \mathbf{y}] &= \mathbf{x}_*^T\boldsymbol{\Sigma}_{w/d}\mathbf{x}_* = \mathbf{x}_*^T\left(\frac{1}{\sigma_r^2}\mathbf{X}\mathbf{X}^T + \boldsymbol{\Sigma}_w^{-1}\right)^{-1}\mathbf{x}_*\end{aligned} \tag{5.57}$$

Furthermore, given a set of M unlabeled test data points in $\mathbf{X}_* = [\mathbf{x}_{1*}, \cdots, \mathbf{x}_{M*}]^T$, we can find

$$\hat{\mathbf{y}}_* = \mathbf{f}_* = \mathbf{f}(\mathbf{x}_*) = \begin{bmatrix} f(\mathbf{x}_{1*}) \\ \vdots \\ f(\mathbf{x}_{M*}) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{1*}^T \\ \vdots \\ \mathbf{x}_{M*}^T \end{bmatrix}\mathbf{w} = [\mathbf{x}_{1*}, \cdots, \mathbf{x}_{M*}]^T\mathbf{w} = \mathbf{X}_*^T\mathbf{w} \tag{5.58}$$

As a linear combination of Gaussian $\mathbf{w}$, $\mathbf{f}_* = \mathbf{X}_*^T\mathbf{w}$ is also Gaussian:

$$p(\mathbf{f}_*|\mathbf{X}_*, \mathcal{D}) = p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(\mathbf{f}_*, \mathbf{X}_*^T\mathbf{m}_{w/d},\ \mathbf{X}_*^T\boldsymbol{\Sigma}_{w/d}\mathbf{X}_*) \tag{5.59}$$

with the following mean and covariance:

$$\mathrm{E}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}] = \mathrm{E}[\mathbf{X}_*^T\mathbf{w}] = \mathbf{X}_*^T\,\mathrm{E}[\mathbf{w}] = \mathbf{X}_*^T\mathbf{m}_{w/d} = \mathbf{X}_*^T\left(\mathbf{X}\mathbf{X}^T + \sigma_r^2\boldsymbol{\Sigma}_w^{-1}\right)^{-1}\mathbf{X}\mathbf{y} \tag{5.60}$$

$$\mathrm{Cov}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}] = \mathrm{E}[\mathbf{f}_*\mathbf{f}_*^T] = \mathrm{E}[\mathbf{X}_*^T\mathbf{w}\mathbf{w}^T\mathbf{X}_*] = \mathbf{X}_*^T\,\mathrm{E}[\mathbf{w}\mathbf{w}^T]\mathbf{X}_* = \mathbf{X}_*^T\boldsymbol{\Sigma}_{w/d}\mathbf{X}_* \tag{5.61}$$

The mean $\mathrm{E}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}]$ above can be considered as the estimated function value $\hat{\mathbf{y}}_* = \mathbf{f}_* = \mathbf{X}_*^T\mathbf{w}$ at the test points in $\mathbf{X}_*$, while the variance $\mathrm{Var}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}]$ as the confidence or certainty of the estimation.

Such a result produced by the Bayesian method in terms of the mean and covariance of $\mathbf{f}_* = \hat{\mathbf{y}}_*$ as a random vector is quite different from the result produced by a frequentist method such as the linear least square regression in terms of an estimated value of $\hat{\mathbf{y}}_*$ as a deterministic variable. However, in the special case of $\sigma_r = 0$, i.e., the residual $\mathbf{r}$ is no longer treated as a random variable, then Eq. (5.60) above becomes the same as Eq. (5.11) produced by linear least

square regression, in other words, the linear method method can be considered as a special case of the Bayesian regression when the residual is deterministic.

The method considered above is based on the linear regression model $f(\mathbf{x}) = \mathbf{x}^T\mathbf{w}$, but it can also be generalized to nonlinear functions, same as in the case of regression based on basis functions discussed in Section 5.3. To do so, the regression function is assumed to be a linear combination of a set of K nonlinear mapping functions $\varphi_k(\mathbf{x})$ $(k = 1, \cdots, K)$:

$$y = f(\mathbf{x}) + e = \sum_{k=1}^{K} w_k\,\varphi_k(\mathbf{x}) + r = \mathbf{w}^T\phi(\mathbf{x}) + r = \phi^T(\mathbf{x})\mathbf{w} + r \tag{5.62}$$

where

$$\mathbf{w} = [w_1, \cdots, w_K]^T, \qquad \phi(\mathbf{x}) = [\varphi_1(\mathbf{x}), \cdots, \varphi_K(\mathbf{x})]^T \tag{5.63}$$

For example, some commonly used mapping functions are listed in Eq. (5.45).

Applying this regression model to each of the N observed data points in $\mathcal{D} = \{(\mathbf{x}_n, y_n),\ (n = 1, \cdots, N)\}$, we get $\hat{y}_n = \sum_{k=1}^{K} w_k\,\varphi_k(x_n)$ with residual $r_n = y_n - \hat{y}_n$, which can be written in matrix form:

$$\begin{aligned}\mathbf{y} &= \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}_{N\times 1} = \begin{bmatrix} \varphi_1(\mathbf{x}_1) & \cdots & \varphi_K(\mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ \varphi_1(\mathbf{x}_N) & \cdots & \varphi_K(\mathbf{x}_N) \end{bmatrix}_{N\times K} \begin{bmatrix} w_1 \\ \vdots \\ w_K \end{bmatrix}_{K\times 1} + \begin{bmatrix} r_1 \\ \vdots \\ r_N \end{bmatrix}_{N\times 1} \\ &= \begin{bmatrix} \phi_1^T \\ \vdots \\ \phi_N^T \end{bmatrix} \mathbf{w} + \mathbf{r} = \mathbf{\Phi}^T\mathbf{w} + \mathbf{r}\end{aligned} \tag{5.64}$$

where, as previously defined in Eq. (5.42),

$$\phi_n = \phi(\mathbf{x}_n) = \begin{bmatrix} \varphi_1(\mathbf{x}_n) \\ \vdots \\ \varphi_K(\mathbf{x}_n) \end{bmatrix}, \qquad \mathbf{\Phi} = [\phi_1, \cdots, \phi_N] \tag{5.65}$$

As this nonlinear model $\hat{\mathbf{y}} = \mathbf{\Phi}^T\mathbf{w}$ is in the same form as the linear model $\hat{\mathbf{y}} = \mathbf{X}^T\mathbf{w}$, all previous discussion for the linear Bayesian regression is still valid for this nonlinear model, if $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]_{(d+1)\times N}$ is replaced by $\mathbf{\Phi} = [\phi_1, \cdots, \phi_N]_{K\times N}$. We can get the covariance matrix and mean vector of the posterior $p(\mathbf{w}|\mathcal{D})$ in the K-dimensional space based on the observed data $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$:

$$\begin{aligned}\mathbf{\Sigma}_{w/d} &= \left(\frac{1}{\sigma_r^2}\mathbf{\Phi}\mathbf{\Phi}^T + \mathbf{\Sigma}_w^{-1}\right)^{-1} \\ \mathbf{m}_{w/d} &= \frac{1}{\sigma_r^2}\mathbf{\Sigma}_{w/d}\mathbf{\Phi}\mathbf{y} = \left(\mathbf{\Phi}\mathbf{\Phi}^T + \sigma_r^2\mathbf{\Sigma}_w^{-1}\right)^{-1}\mathbf{\Phi}\mathbf{y}\end{aligned} \tag{5.66}$$

The function $\hat{\mathbf{y}}_* = \mathbf{f}(\mathbf{X}_*)$ of the multiple test points in $\mathbf{X}_* = [\mathbf{x}_{*1}, \cdots, \mathbf{x}_{*M}]^T$ is

a linear transformation of $\mathbf{w}$:

$$\hat{\mathbf{y}}_* = \mathbf{f}_* = \mathbf{f}(\mathbf{x}_*) = \begin{bmatrix} f_1(\mathbf{x}_*) \\ \vdots \\ f_M(\mathbf{x}_*) \end{bmatrix} = \begin{bmatrix} \phi_1^T \\ \vdots \\ \phi_M^T \end{bmatrix}_* \mathbf{w} = [\phi_1, \cdots, \phi_M]^T \mathbf{w} = \mathbf{\Phi}_*^T \mathbf{w} \tag{5.67}$$

and it also has a Gaussian joint distribution:

$$p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(\mathbf{\Phi}_*^T \mathbf{m}_{w/d},\ \mathbf{\Phi}_*^T \mathbf{\Sigma}_{w/d} \mathbf{\Phi}_*) \tag{5.68}$$

with the mean and covariance:

$$\begin{aligned} \mathrm{E}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}] &= \mathrm{E}[\mathbf{\Phi}_*^T \mathbf{w}] = \mathbf{\Phi}_*^T \mathrm{E}[\mathbf{w}] = \mathbf{\Phi}_*^T \mathbf{m}_{w/d} \\ \mathrm{Cov}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}] &= \mathrm{E}[\mathbf{f}_* \mathbf{f}_*^T] = \mathrm{E}[\mathbf{\Phi}_*^T \mathbf{w}\mathbf{w}^T \mathbf{\Phi}_*] \\ &= \mathbf{\Phi}_*^T \mathrm{E}[\mathbf{w}\mathbf{w}^T] \mathbf{\Phi}_* = \mathbf{\Phi}_*^T \mathbf{\Sigma}_{w/d} \mathbf{\Phi}_* \end{aligned} \tag{5.69}$$

They can be reformulated to become:

$$\begin{aligned} \mathrm{E}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}] &= \mathbf{\Phi}_*^T \mathbf{m}_{w/d} = \mathbf{\Phi}_*^T \left(\mathbf{\Phi}\mathbf{\Phi}^T + \sigma_r^2 \mathbf{\Sigma}_w^{-1} \right)^{-1} \mathbf{\Phi}\mathbf{y} \\ &= \mathbf{\Phi}_*^T (\mathbf{\Sigma}_w \mathbf{\Phi})(\mathbf{\Sigma}_w \mathbf{\Phi})^{-1} \left(\mathbf{\Phi}\mathbf{\Phi}^T + \sigma_r^2 \mathbf{\Sigma}_w^{-1} \right)^{-1} (\mathbf{\Phi}^{-1})^{-1} \mathbf{y} \\ &= \mathbf{\Phi}_*^T \mathbf{\Sigma}_w \mathbf{\Phi} \left[\mathbf{\Phi}^{-1} \left(\mathbf{\Phi}\mathbf{\Phi}^T + \sigma_r^2 \mathbf{\Sigma}_w^{-1} \right) \mathbf{\Sigma}_w \mathbf{\Phi} \right]^{-1} \mathbf{y} \\ &= \mathbf{\Phi}_*^T \mathbf{\Sigma}_w \mathbf{\Phi} (\mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi} + \sigma_r^2 \mathbf{I})^{-1} \mathbf{y} \end{aligned} \tag{5.70}$$

and

$$\begin{aligned} \mathrm{Cov}[\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}] &= \mathbf{\Phi}_*^T \mathbf{\Sigma}_{w/d} \mathbf{\Phi}_* = \mathbf{\Phi}_*^T \left(\frac{1}{\sigma_r^2} \mathbf{\Phi}\mathbf{\Phi}^T + \mathbf{\Sigma}_w^{-1} \right)^{-1} \mathbf{\Phi}_* \\ &= \mathbf{\Phi}_*^T \left[\mathbf{\Sigma}_w - \mathbf{\Sigma}_w \mathbf{\Phi} (\mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi} + \sigma_r^2 \mathbf{I})^{-1} \mathbf{\Phi}^T \mathbf{\Sigma}_w \right] \mathbf{\Phi}_* \\ &= \mathbf{\Phi}_*^T \mathbf{\Sigma}_w \mathbf{\Phi}_* - \mathbf{\Phi}_*^T \mathbf{\Sigma}_w \mathbf{\Phi} (\mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi} + \sigma_r^2 \mathbf{I})^{-1} \mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi}_* \end{aligned} \tag{5.71}$$

where we have used Woodbury identity in Section A.2.4.

We note that in these expressions the mapping function $\varphi(\mathbf{x})$ only shows up in the quadratic form of $\mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi}$, $\mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi}_*$, or $\mathbf{\Phi}_*^T \mathbf{\Sigma}_w \mathbf{\Phi}_*$. This suggests that we may consider using *kernel method* (Section 14.2, by which some significant advantage may be gained (such as in the support-vector machines (Section 14). Specifically, if we replace the mn-th component $\phi^T(\mathbf{x}_m) \mathbf{\Sigma}_w \phi(\mathbf{x}_n)$ of $\mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi}$ by a *kernel function* only in terms of $\mathbf{x}_m$ and $\mathbf{x}_n$:

$$\phi^T(\mathbf{x}_m) \mathbf{\Sigma}_w \phi(\mathbf{x}_n) = k(\mathbf{x}_m, \mathbf{x}_n) = k(\mathbf{x}_n, \mathbf{x}_m) \tag{5.72}$$

then the mapping function $\varphi(\mathbf{x})$ no longer needs to be explicitly specified. This idea is called the *kernel trick*, Now we have

$$\mathbf{\Phi}^T \mathbf{\Sigma}_w \mathbf{\Phi} = \begin{bmatrix} \phi_1^T \\ \vdots \\ \phi_N \end{bmatrix} \mathbf{\Sigma}_w \left[\phi_1, \cdots, \phi_N \right] = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \cdots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix} = \mathbf{K} \tag{5.73}$$

and similarly we also have $\mathbf{\Phi}^T\mathbf{\Sigma}_w\mathbf{\Phi}_* = \mathbf{K}_*$ and $\mathbf{\Phi}_*^T\mathbf{\Sigma}_w\mathbf{\Phi} = \mathbf{K}_*^T$. These kernel functions are symmetric, same as all covariance matrices. Now the mean and covariance of $\mathbf{f}_*$ given above can be expressed as:

$$\begin{aligned} \mathrm{E}[\mathbf{f}_*] &= \mathbf{K}_*^T \left(\mathbf{K} + \sigma_r^2\mathbf{I}\right)^{-1}\mathbf{y} \\ \mathrm{Cov}[\mathbf{f}_*] &= \mathbf{K}_{**} - \mathbf{K}_*^T \left(\mathbf{K} + \sigma_r^2\mathbf{I}\right)^{-1}\mathbf{K}_* \end{aligned} \tag{5.74}$$

As a general summary of the method discussed above, we note that it is essentially different from the frequentist approach, in the sense that the estimated result produced is no longer a set specific values of some unknown variables in question (function values, parameters of regression models, etc.), but the probability distribution of these variables in terms of the expectation and covariance. Consequently, we can not only treat the expectation as the estimated values, but also use the variance as the confidence or certainty of such an estimation.

Below is a segment of Matlab code for estimating the mean and covariance of weight vector $\mathbf{w}$ by the Bayes linear regression. Here `X` and `y` are for $\mathbf{X}$ and $\mathbf{y}$ of the training data $\mathcal{D}$, and `Xstar`, `Phi` and `Phistar` are for $\mathbf{X}_*$, $\mathbf{\Phi}$ and $\mathbf{\Phi}_*$, respectively.

```
SigmaP=eye(d);   % prior variance of r in original space
SigmaQ=eye(D);   % prior variance of r in feature space
Sigmaw1=inv(X*X'/sigmaN^2+inv(SigmaP));
                 % covariance of w by Bayes regression
Sigmaw2=inv(Phi0*Phi0'/sigmaN^2+inv(SigmaQ));
                 % cov of w with basis function

w0=inv(X*X')*X*y % weight vector w by LS method
w1=Sigmaw1*X*y/sigmaN^2 % mean of w by Bayes regression
w2=Sigmaw2*Phi0*y/sigmaN^2;
                 % mean of w by Bayes regression with basis function

y0=w0'*Xstar;    % y by least squares method
y1=w1'*Xstar;    % y by Bayes regression
y2=w2'*Phistar0; % y by Bayes regression with basis function

for i=1:M
    sgm1(i)=sqrt(Xstar(:,i)'*Sigmaw1*Xstar(:,i));
                   % variance of estimated y
    sgm2(i)=sqrt(Phistar(:,i)'*Sigmaw2*Phistar(:,i));
end
```

Example 5.11 Consider a 2-D linear function

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = [w_0,\, w_1]\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = w_0x_0 + w_1x_1 = 0.5 + 0.5\,x_1 \tag{5.75}$$

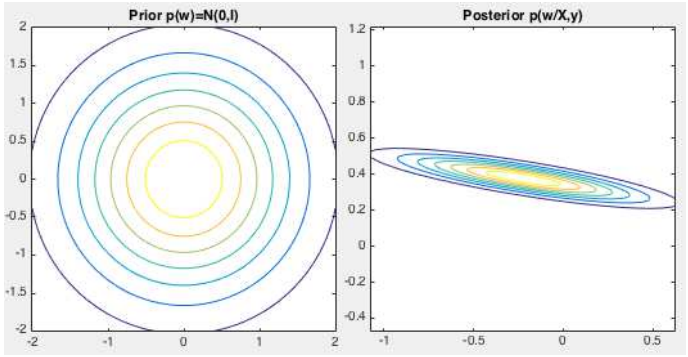


Figure 5.24 Prior and Posterior of Bayesian Regression

where we have assumed $x_0 = 1$, $w_0 = -1/2$ and $w_1 = 1/2$, i.e., the function is actually a straight line $f(x) = (-1 + x)/2$ with intercept $w_0 = -1/2$ and slope $w_1 = 1/2$. The distribution of the additive noise (residual) is $p(e) = \mathcal{N}(0, \sigma_r = 0.8)$. The prior distributions of the weights $\mathbf{w} = [w_0, w_1]^T$ is assumed to be $p(\mathbf{w}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, and posterior $p(\mathbf{w}|\mathcal{D})$ can be obtained based on the observed data $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$, both shown in Fig. 5.24. We see that the posterior is concentrated around $\hat{w}_1 = -0.28$ and $\hat{w}_2 = 0.39$, the estimates of the true weights $w_1 = -0.5$ and $w_2 = 0.5$, based on the observed noisy data.

Various regression methods including both the least-squared method and the Bayesian linear regression methods, and in feature space spanned by a set of basis functions $\phi_i(\mathbf{x})$, $(i = 1, \cdots, D = 5)$ based on each of the three functions given above are carried out with their results shown in Fig. 5.25.

The LS errors for these methods are listed below:

Method	LS Error	$\mathbf{w} = [w_1, w_2]^T$
Pseudo inverse	0.4730	$[-0.28, 0.39]^T$
Bayesian linear regression	0.4742	$[-0.22, 0.38]^T$
$\phi = \exp(a\|\|\mathbf{x} - \mathbf{c}\|\|^2)$	0.6196	
$\phi = \|\|\mathbf{x} - \mathbf{c}\|\|$	0.4429	
$\phi = \exp(-a\|\|\mathbf{x} - \mathbf{c}\|\|^2)$	0.4459	

(5.76)

We see that the linear methods (the pseudo inverse and the Bayesian regression) produce similar estimated weights error, but smaller error can be achieved by the same method of Bayesian linear regression if proper nonlinear mapping is used (e.g., the last two mapping functions).

Problems

1. Develop Matlab code for both left and right pseudo inverse (Section A.6.1) of an $m \times n$ matrix $\mathbf{A}$ ($m > n$ or $m < n$).

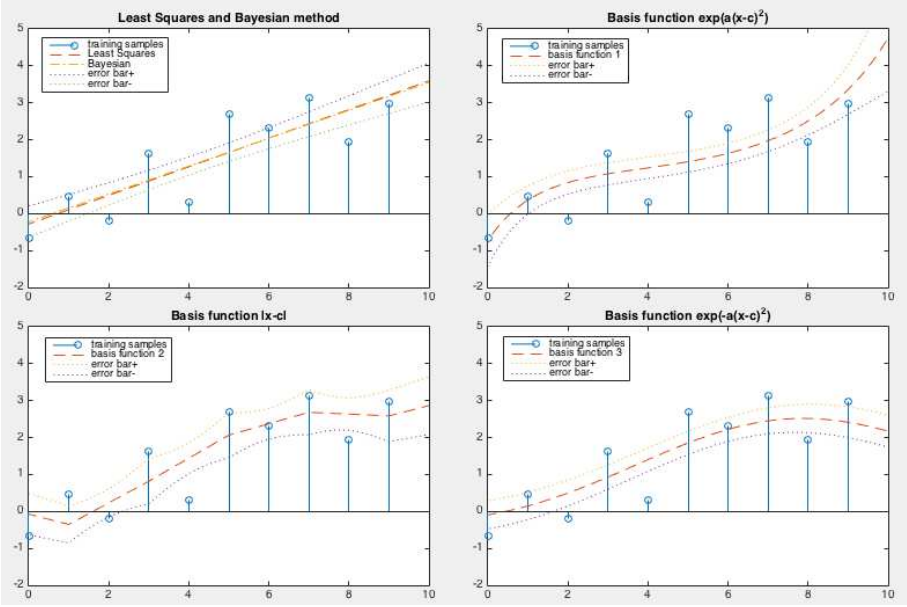


Figure 5.25 Bayesian Regression (Linear and Basis Function Based)

2. Solve the over-constrained linear equation system $\mathbf{Ax} = \mathbf{b}$ and find the squared error, with the coefficient matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 3 & -2 \\ 1 & 1 \end{bmatrix}$$

and each of the two different right-hand sides:

$$\mathbf{b}_1 = \begin{bmatrix} 8 \\ -1 \\ 3 \end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix} 8 \\ 1 \\ 3 \end{bmatrix}$$

3. Develop Matlab code to solve ordinary least squares regression problems to find the $d+1$ dimensional weight vector $\mathbf{w} = [w_0 = b, w_1, \cdots, w_d]^T$, and also other associated parameters including the sum of squares ESS, RSS, TSS, R^2, as well as the correlation coefficient ρ.

   Apply your code to the following two data sets, find $\mathbf{w}$ and all parameters above, and then plot the regression curve $y = w_0 + w_1 x$ together with the data points $\{(x_i, y_i)\, i = 1, \cdots, N\}$.

   - 2-D dataset in/e176/programs/dataLS1.txt, containing two columns for (x, y) of each data point.
   - 3-D dataset in /e176/programs/dataLS2.txt, containing three columns for (x_1, x_2, y) of each data point.

4. Implement ridge regression and apply it to a set of synthetic 2-D data (generated by your own code) including an independent variable x_1 and another closely correlated variable x_2 (a noisy version of the x_1, by, for example, `x2=x1+a*rand(1,N)`). Try different noise level by varying `a` see how ill-behaved the dataset is in terms of correlation coefficient ρ between the two variables, and how ill-conditioned the problem is in terms of how close to zero one of the eigenvalues of matrix $\mathbf{X}\mathbf{X}^T$ is. Try different λ values of the ridge regression (including $\lambda = 0$ for regular linear regression) to see how it affects the stability of the resulting coefficients $w_0,\ w_1,\ w_2$ and the accuracy in terms of R^2.
5. Implement the method of nonlinear regression based on each of four different basis functions considered in Section 5.3. Apply your code to solve the same 1-D regression problems given in Example 5.8. Compare results with different K values to see how the number of basis functions affects the results.
6. Develop Matlab code to implement the method of nonlinear regression based on each of the three basis functions, Gaussian, ReLU, and sigmoid considered in Section 5.3. Apply the code to solve the same 2-D regression problems given in Examples 5.9 and 5.10.
7. Develop Matlab code to implement the method of Bayesian regression based on different basis functions. Apply the code to the same 1-D problem in Example 5.11, and compare results with that of the LS method.
8. Apply the code developed in the previous problem to the same 2-D regression problem given in Examples 5.9 and 5.10.

6 Nonlinear Regression

6.1 Nonlinear Least Squares Regression

The goal of nonlinear least squares regression is to model the nonlinear relationship between the dependent variable y and independent variables in $\mathbf{x} = [x_1, \cdots, x_d]^T$, based on the given training dataset $\mathcal{D} = \{(\mathbf{x}_n, y_n),\ n = 1, \cdots, N\}$, by a nonlinear regression function parameterized by $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_M]^T$:

$$\hat{y} = f(x_1, \cdots, x_d,\ \theta_1, \cdots, \theta_M) = f(\mathbf{x}, \boldsymbol{\theta}) \tag{6.1}$$

While the specific form of the nonlinear function is typically determined based on certain prior knowledge, we need to find the parameter $\boldsymbol{\theta}$ for the model to fit the training dataset $\mathcal{D}$, so that, its output $\hat{\mathbf{y}}$ matches the ground truth labeling $\mathbf{y}$ in the training set, i.e., ideally the residual $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}}$ below is zero:

$$\mathbf{r}(\boldsymbol{\theta}) = \begin{bmatrix} r_1 \\ \vdots \\ r_N \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} f_1(\boldsymbol{\theta}) \\ \vdots \\ f_N(\boldsymbol{\theta}) \end{bmatrix} = \mathbf{y} - \hat{\mathbf{y}}(\boldsymbol{\theta}) = \mathbf{y} - \mathbf{f}(\mathbf{X}, \boldsymbol{\theta}) = \mathbf{0} \tag{6.2}$$

where $\hat{y}(\mathbf{x}_n, \boldsymbol{\theta}) = f_n(\boldsymbol{\theta}) = f(\mathbf{x}_n, \boldsymbol{\theta})$ is the output generated by the regression function based on the nth data point $\mathbf{x}_n$. We see that the regression problem is equivalent to solving a nonlinear equation system of N equations for the M unknown parameters in $\boldsymbol{\theta}$. As typically there are many more data samples than the unknown parameters, i.e., $N \gg M$, the equation $\mathbf{r} = \mathbf{0}$ is over-overconstrained without an exact solution. We need to find the the optimal parameter $\boldsymbol{\theta}^*$ that minimizes the sum of squared error (SSE) based on the residual $\mathbf{r}$:

$$\varepsilon(\boldsymbol{\theta}) = \frac{1}{2}||\mathbf{r}||^2 = \frac{1}{2}\sum_{n=1}^{N} r_n^2 = \frac{1}{2}\sum_{n=1}^{N}[y_n - f_n(\boldsymbol{\theta})]^2 \tag{6.3}$$

This optimization problem can be addressed by either the gradient descent methods or the Gauss-Newton method.

We first consider the gradient descent method based on the gradient of this

error function:

$$\mathbf{g}_\varepsilon(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\varepsilon(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\left(\frac{1}{2}\sum_{n=1}^{N} r_n^2\right) = \sum_{n=1}^{N}\frac{d\, r_n}{d\boldsymbol{\theta}}\, r_n$$

$$= -\sum_{n=1}^{N}\frac{d}{d\boldsymbol{\theta}} f_n(\boldsymbol{\theta})[y_n - f_n(\boldsymbol{\theta})] = -\mathbf{J}_f^T\, \mathbf{r} \tag{6.4}$$

where $\mathbf{J}_f$ is the $N \times M$ Jacobian matrix $\mathbf{J}_f$ of $\mathbf{f}(\mathbf{X}, \boldsymbol{\theta})$. We can also obtain the mth $(m = 1, \cdots, M)$ component of the gradient vector as:

$$g_m(\boldsymbol{\theta}) = \frac{\partial}{\partial\theta_m}\varepsilon(\boldsymbol{\theta}) = \frac{1}{2}\frac{\partial}{\partial\theta_m}||\mathbf{r}||^2 = \frac{1}{2}\sum_{n=1}^{N}\frac{\partial}{\partial\theta_m} r_n^2 = \sum_{n=1}^{N}\frac{\partial r_n}{\partial\theta_m}\, r_n$$

$$= -\sum_{n=1}^{N}\frac{\partial f_n(\boldsymbol{\theta})}{\partial\theta_m}[y_n - f_n(\boldsymbol{\theta})] = -\sum_{n=1}^{N} J_{nm}\,[y_n - f_n(\boldsymbol{\theta})] = -\sum_{n=1}^{N} J_{nm}\, r_n \tag{6.5}$$

where $J_{nm} = \partial\, f_n(\boldsymbol{\theta})/\partial\theta_m$ is the component in the nth row and mth column of $\mathbf{J}_f$, which is simply the negative version of Jacobian of $\mathbf{r} = \mathbf{y} - \mathbf{f}(\mathbf{X}, \boldsymbol{\theta})$:

$$\mathbf{J}_r(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\mathbf{r}(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}[\mathbf{y} - \mathbf{f}(\mathbf{X}, \boldsymbol{\theta})] = -\frac{d}{d\boldsymbol{\theta}}\mathbf{f}(\mathbf{X}, \boldsymbol{\theta}) = -\mathbf{J}_f(\boldsymbol{\theta}) \tag{6.6}$$

The optimal parameter $\boldsymbol{\theta}$ that minimizes $\varepsilon(\boldsymbol{\theta})$ can now be found iteratively by the gradient descent method

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \delta_n \mathbf{g}_\varepsilon(\boldsymbol{\theta}_n) = \boldsymbol{\theta}_n + \delta_n \mathbf{J}_f^T\, \mathbf{r} \tag{6.7}$$

Setting the value for the step size δ_n properly may be tricky, as it depends on the specific landscape of the error hypersurface $\varepsilon(\boldsymbol{\theta})$. If δ_n is too large, a minimum may be skipped over, but if δ_n is too large, the convergence of the iteration may be too slow.

This gradient method is also called *batch gradient descent (BGD)*, as the gradient in Eq. (6.4) is calculated based on the entire batch of all N data points in the dataset $\mathcal{D}$ in each iteration. When the size of the dataset is large, BGD may be slow due to the high computational complexity. In this case the method of *stochastic gradient descent (SGD)* can be used, which approximates the gradient based on only one of the N data points uniformly randomly chosen from the dataset in each iteration, i.e., the summation in Eq. (6.4) contains only one term corresponding to the chosen sample $\mathbf{x}_n$:

$$\mathbf{g}_n(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\frac{1}{2} r_n^2 = r_n \frac{d\, r_n}{d\boldsymbol{\theta}} \tag{6.8}$$

The expectation of $\mathbf{g}_n$ based on one data point is the same as the true gradient (scaled by $1/N$) based on all N data points in the dataset

$$E[\mathbf{g}_n(\boldsymbol{\theta})] = \sum_{n=1}^{N} P(\mathbf{x}_n)\mathbf{g}_n(\boldsymbol{\theta}) = \sum_{n=1}^{N}\frac{1}{N}\mathbf{g}_n(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}\frac{dr_n}{d\boldsymbol{\theta}}\, r_n = \frac{1}{N}\mathbf{g}(\boldsymbol{\theta}) \tag{6.9}$$

As the data are typically noisy, the SGD gradient based on only one data point

may be in a direction very different from the true gradient based on the mean of all N data points, and the resulting search path in the M-D space may be in a very jagged zig-zag shape, and the iteration may converge slowly. However, as the computational complexity in each iteration is much reduced, we can afford to carry out a large number of iterations with an overall search path generally coinciding with that of the BGD method.

Some tradeoff between BGD and SGD can be made so that the gradient is estimated based on more than one but less than all N data points in the dataset, the resulting method, called *mini-batch gradient descent*, can take advantage of both methods. The methods of stochastic and mini-batch gradient descent are widely used in machine learning, such as in the method of artificial neural networks, which can be treated as a nonlinear regression problem, considered in Part V.

We then consider the Gauss-Newton method to find the optimal parameter $\boldsymbol{\theta}^*$ of the nonlinear regression function $f(\mathbf{x}, \boldsymbol{\theta})$ by directly solving the nonlinear over-determined equation system $\mathbf{r}(\boldsymbol{\theta}) = \mathbf{y} - \mathbf{f}(\mathbf{X}, \boldsymbol{\theta}) = \mathbf{0}$ in Eq. (6.2), based on the iteration in Eq. (1.77) in Chapter 1:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \mathbf{J}_r^- \, \mathbf{r} = \boldsymbol{\theta}_n - (\mathbf{J}_r^T \mathbf{J}_r)^{-1} \mathbf{J}_r^T \, \mathbf{r} = \boldsymbol{\theta}_n + (\mathbf{J}_f^T \mathbf{J}_f)^{-1} \mathbf{J}_f^T \, \mathbf{r} \tag{6.10}$$

The last equality is due to the fact that $\mathbf{J}_r = -\mathbf{J}_f$. Here $\mathbf{J}_f^T \mathbf{J}_f$ in the pseudo-inverse is actually an approximation of the Hessian $\mathbf{H}_\varepsilon$ of the error function $\varepsilon(\boldsymbol{\theta})$, as given in Eq. (2.20) in Chapter 2, and $-\mathbf{J}_f^T \mathbf{r} = \mathbf{g}_\varepsilon(\boldsymbol{\theta})$ is the gradient of the error function given in Eq. (6.4). So the iteration above is essentially:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \mathbf{J}_r^- \, \mathbf{r} \approx \boldsymbol{\theta}_n - \mathbf{H}_\varepsilon^{-1}(\boldsymbol{\theta}_n) \mathbf{g}_\varepsilon(\boldsymbol{\theta}_n) \tag{6.11}$$

same as the Newton-Raphson method based on the second order Hessian as well as the first order gradient. We therefore see that the Gauss-Newton method is actually for minimizing the sum of squares error is essentially the same as the Newton-Raphson method, with the only difference that the Hessian $\mathbf{H}_\varepsilon \approx \mathbf{J}_f^T \mathbf{J}_f$ as the second order derivative of the error function $\varepsilon(\boldsymbol{\theta})$ is approximated by the Jacobian $\mathbf{J}_f \boldsymbol{\theta}$ as the first order derivative of the regression function $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$, making the implementation computationally easier.

In the special case of linear regression function $\mathbf{f}(\mathbf{w}) = \mathbf{x}^T \mathbf{w}$, we have:

$$\hat{\mathbf{y}} = \mathbf{f}(\mathbf{X}, \mathbf{w}) = \begin{bmatrix} f(\mathbf{x}_1, \mathbf{w}) \\ \vdots \\ f(\mathbf{x}_N, \mathbf{w}) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \mathbf{w} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]^T \mathbf{w} = \mathbf{X}^T \mathbf{w} \tag{6.12}$$

where $\mathbf{w} = [w_0, w_1, \cdots, w_d]^T$ and $\mathbf{x}_i = [1, x_{i1}, \cdots, x_{id}]^T$ are both augmented $d+1$ dimensional vectors and $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$. The Taylor series is

$$f(\mathbf{X}, \mathbf{w} + \Delta\mathbf{w}) = \mathbf{X}^T(\mathbf{w} + \Delta\mathbf{w}) = \mathbf{X}^T\mathbf{w} + \mathbf{X}^T\Delta\mathbf{w} \tag{6.13}$$

and the Jacobian is

$$\mathbf{J}_f(\mathbf{w}) = \mathbf{X}^T \tag{6.14}$$

Now Eq. (6.10) becomes

$$\begin{aligned}\mathbf{w}_{n+1} &= \mathbf{w}_n + (\mathbf{X}^T)^-(\mathbf{y} - \mathbf{X}^T\mathbf{w}_n) = \mathbf{w}_n + (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}(\mathbf{y} - \mathbf{X}^T\mathbf{w}_n) \\ &= \mathbf{w}_n + (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y} - (\mathbf{X}\mathbf{X}^T)^{-1}(\mathbf{X}\mathbf{X}^T)\mathbf{w}_n \\ &= (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y} = (\mathbf{X}^T)^-\mathbf{y} \end{aligned} \tag{6.15}$$

i.e., the optimal parameter $\mathbf{w}^*$ can be found in a single step without iteration, same as that in Eq. (5.10).

The Matlab code for the essential parts of the algorithm is listed below, where `data(:,1)` and `data(:,2)` are the observed data containing N samples $x_1, \cdots, x_N$ in the first column and their corresponding function values $y_1, \cdots, y_N$ in the second column. In the code segment below, the regression function (e.g., an exponential model) is symbolically defined, and then a function `NLregression` is called to estimate the function parameters:

```
sym x;                          % symbolic variables
a=sym('a',[3 1]);               % symbolic parameters
f=@(x,a)a(1)*exp(a(2)*x)+a(3)|    % symbolic function
[a er]=NLregression(data(:,1),data(:,2),f,3);
                                % call regression
```

The code below estimates the M parameters of the regression function based on Newton's method.

```
function [a er]=NLregression(x,y,f,M)   % nonlinear regression
    N=length(x);                % number of data samples
    a=sym('a',[M 1]);           % M symbolic parameters
    F=[];
    for i=1:N                   % for N given points
        F=[F; f(x(i),a)];       % evaluate function at each point
    end
    J=jacobian(F,a);            % symbolic Jacobian
    J=matlabFunction(J);        % convert to a true function
    F=matlabFunction(F);
    a1=ones(M,1);               % initial guesses for M parameters
    n=0;                        % iteration index
    da=1;
    while da>tol                % terminate if no progress made
        n=n+1;
        a0=a1;                  % update parameters
        J0=J(a0(1),a0(2));      % evaluate Jacobian based on a0
        Jinv=pinv(J0);          % pseudo inverse of Jacobian
        r=y-F(a0(1),a0(2))      % residual
        a1=a0+Jinv*r;           % Newton iteration
        er=norm(r);             % residual error
        da=norm(a0-a1);         % progress
```

```
    end
end
```

where `tol` is a small value for tolerance, such as 10^{-6}.

Example 6.1 Consider three sets of data (shown as the solid dots in the three columns in the plots in Fig. 6.1) are fitted by three different models (shown as the open circles in the plots):

1. Linear model: $y = f_1(x, a, b) = ax + b$
2. quadratic model: $y = f_2(x, a, b, c) = ax^2 + bx + c$
3. Exponential model: $y = f_3(x, a, b, c) = a\, e^{bx} + c$

These are the parameters for the three models to fit each of the three datasets:

	Parameters :	a	b	c	*error*
	Model 1 :	30.337	51.652		7.4294
Dataset 1	*Model* 2 :	0.076	30.034	51.580	7.4223
	Model 3 :	5933.214	0.005	−5881.634	7.4221
	Model 1 :	24.519	20.089		21.0362
Dataset 2	*Model* 2 :	4.459	6.625	15.855	8.4559
	Model 3 :	29.791	0.346	−10.202	9.7402
	Model 1 :	32.608	22.008		53.0724
Dataset 3	*Model* 2 :	10.391	−9.089	12.140	28.3211
	Model 3 :	2.197	0.885	27.241	8.3393

We see that the first dataset can be fitted by all three models equally well with similar error (when $a \approx 0$ in model 2, it is approximately a linear functions), the second dataset can be well fitted by either the second or the third model, while the third dataset can only be fitted by the third exponential model.

Example 6.2 Based on the following three functions with four parameters $a = 1,\ b = 2,\ c = 3,\ d = 4$, three sets of synthetic 2-D data points are generated with some random noise added, shown in Fig. 6.2 as the dots in the three rows.

$$\begin{aligned} f_1(\mathbf{x}) &= ax_1 + bx_2 - c \\ f_2(\mathbf{x}) &= \exp(ax_1 + bx_2) - cx_1x_2 + d \\ f_3(\mathbf{x}) &= ax_1^2 - bx_1x_2 + cx_2^2 + d \end{aligned} \tag{6.16}$$

Then each of three datasets are fitted by the three functions as models while the optimal parameters are found by the Gauss-Newton method.

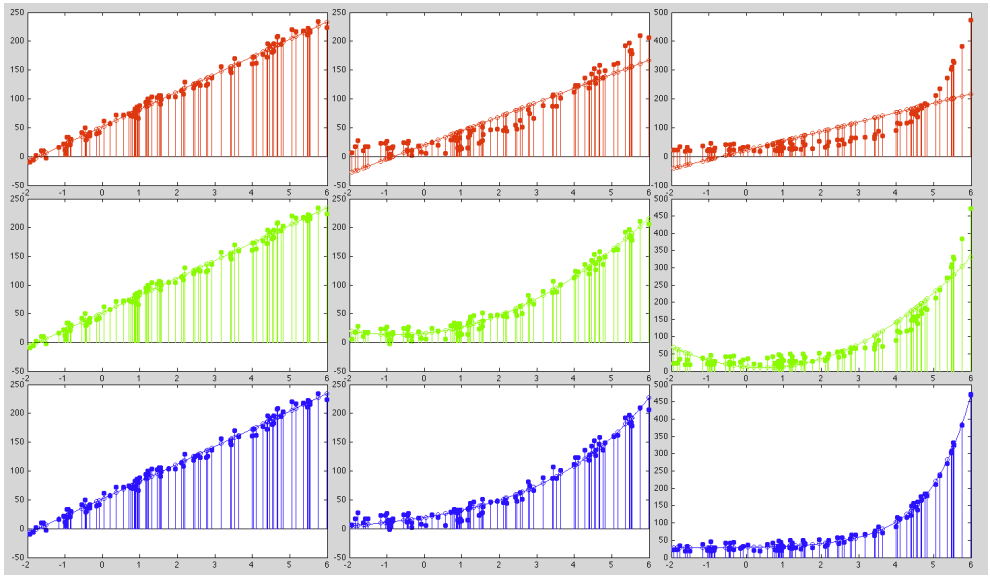


Figure 6.1 Nonlinear Regression with Multiple Datasets and Models (1-D)

	Parameters :	a	b	c	d	*error*
Dataset 1	*Model* 1 :	0.98	1.97	3.00		4.13
	Model 2 :	0.70	1.30	1.21	−4.42	6.87
	Model 3 :	−0.19	0.73	−0.12	−3.01	18.03
Dataset 2	*Model* 1 :	1.84	3.40	−6.09		14.70
	Model 2 :	0.99	2.01	2.97	3.98	3.75
	Model 3 :	0.37	1.13	2.66	4.87	32.45
Dataset 3	*Model* 1 :	−0.19	0.30	−5.34		16.82
	Model 2 :	−0.29	0.23	2.00	4.30	13.51
	Model 3 :	0.94	2.05	2.99	4.02	4.09

The *Levenberg-Marquardt algorithm (LMA)* or *damped least-squares method* is a method widely used to minimize a sum of squares error ε as in Eq. (6.3). The LMA is an interpolation or tradeoff between the the gradient descent method based on the first order derivative of the error function to be minimized, and the Gauss-Newton method based on both the first and second order derivatives of the error function, both discussed above. As the LMA can take advantage of both methods and is therefore more robust. Even if the initial guess is far from the solution corresponding to the minimum of the objective function, the iteration can still converge toward the solution.

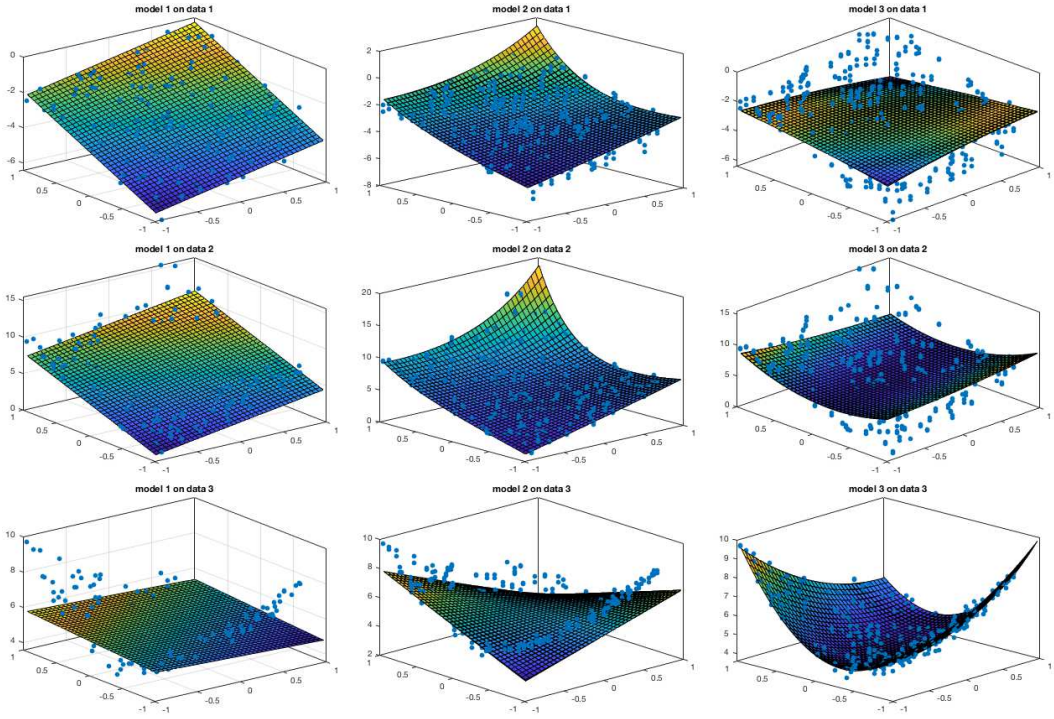


Figure 6.2 Nonlinear Regression with Multiple Datasets and Models (2-D)

Specifically, the LMA modifies Eq. (6.10)

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \mathbf{J}_f^-(\boldsymbol{\theta}_n)[\mathbf{y} - \mathbf{f}(\mathbf{X}, \boldsymbol{\theta}_n)] = \boldsymbol{\theta}_n + (\mathbf{J}_n^T\mathbf{J}_n)^{-1}\mathbf{J}_n^T[\mathbf{y} - \mathbf{f}(\mathbf{X}, \boldsymbol{\theta}_n)] \quad (6.17)$$

to include an extra term proportional to the identity matrix $\mathbf{I}$:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + (\mathbf{J}_n^T\mathbf{J}_n + \lambda\mathbf{I})^{-1}\mathbf{J}_n^T[\mathbf{y} - \mathbf{f}(\mathbf{X}, \boldsymbol{\theta}_n)] \quad (6.18)$$

where $\lambda \geq 0$ is the non-negative damping factor, which is to be adjusted at each iteration. We recognize that this is actually similar to Eq. (5.34) for ridge regression.

In each iterative step, if λ is small and the error function $\varepsilon(\boldsymbol{\theta}_n)$ decreases quickly, indicating the Gauss-Newton is effective in the sense that the error function is well modeled as a quadratic function by the first and second order terms of the Taylor series, we will keep λ small. On the other hand, if ε decreases slowly, indicating the error function may not be well modeled as a quadratic function, due possibly to the fact that the current $\boldsymbol{\theta}_n$ is far away from the minimum, then we let λ take a large value, so that the term $\mathbf{J}_n^T\mathbf{J}_n$ becomes insignificant, and Eq. (6.18) approaches

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \lambda\mathbf{J}_n^T(\mathbf{y} - \mathbf{f}(\boldsymbol{\theta})) = \boldsymbol{\theta}_n - \alpha\mathbf{g}_\varepsilon(\boldsymbol{\theta}) \quad (6.19)$$

Now $\boldsymbol{\theta}_n$ is updated along the negative direction of the gradient in Eq. (6.4):

$$\mathbf{g}_\varepsilon(\boldsymbol{\theta}) \propto \mathbf{J}^T(\mathbf{f}(\boldsymbol{\theta}) - \mathbf{y}) \tag{6.20}$$

with some step size α.

By adjusting the parameter λ, the LMA can switch between either of the two methods for the same purpose of minimizing $\varepsilon(\boldsymbol{\theta})$. When the value of λ is small, the iteration is dominated by the Gauss-Newton method that approximate the error function by a quadratic function based on the second order derivatives. If the error reduces slowly, indicating the error function cannot be approximated by a quadratic function, a greater value of λ is used to switch to the gradient descent method.

6.2 Parameter Estimation by Natural Gradient Descent

In Section 4.1, we considered the MLE and MAP methods for estimating the parameter $\boldsymbol{\theta}$ of a probability model $p(\mathbf{x}|\boldsymbol{\theta})$ by maximizing the likelihood function $L(\boldsymbol{\theta}|\mathbf{x})$ or the posterior $p(\boldsymbol{\theta}|\mathbf{x})$. In the previous section we also considered how the gradient method is used to estimate the model parameter $\boldsymbol{\theta}$ by minimizing the squared error. Obviously we can also use the gradient method in MLE to maximize the likelihood. However, unlike the parameter $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_m]^T$ which is treated as a vector in an m-D Euclidean space spanned by a set of orthonormal basis vectors, the family of distributions $p(\mathbf{x}|\boldsymbol{\theta})$ corresponding to all possible parameters $\boldsymbol{\theta}$ resides in a distribution space, an m-dimensional Riemannian manifold or space, in which no orthogonal basis is defined, and methods based on concepts such as inner product and Euclidean distance are no longer valid.

In this case, the difference between any two distributions $p(\mathbf{x}|\boldsymbol{\theta})$ and $p(\mathbf{x}|\boldsymbol{\theta}')$ can be measured by their KL-divergence $KL(p(\mathbf{x}|\boldsymbol{\theta})||p_p(\mathbf{x}|\boldsymbol{\theta} + \Delta\boldsymbol{\theta}))$, and the gradient method can still be used to find iteratively the optimal parameter that maximize the likelihood $L(\boldsymbol{\theta})$, similar to the steps that lead to Eq. (3.32). Specifically, we desired to find the optimal $\Delta\boldsymbol{\theta}$ that maximally increases the likelihood, which is also subject to the constraint $||\Delta\boldsymbol{\theta}||^2 < C$:

$$\Delta\boldsymbol{\theta}^* = \underset{||\Delta\boldsymbol{\theta}||^2 < C}{\arg\max}\, L(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \tag{6.21}$$

where $||\Delta\boldsymbol{\theta}||^2$ is defined as Eq. (B.275):

$$||\Delta\boldsymbol{\theta}||^2 = \frac{1}{2}\Delta\boldsymbol{\theta}^T\mathbf{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} = KL(p(\mathbf{x}|\boldsymbol{\theta})||p_p(\mathbf{x}|\boldsymbol{\theta} + \Delta\boldsymbol{\theta})) < C \tag{6.22}$$

To solve this constrained optimization problem we set the derivative of the Lagrangian to zero

$$\frac{d}{d(\Delta\boldsymbol{\theta})}\left[L(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) + \mu\left(\frac{1}{2}\Delta\boldsymbol{\theta}^T\mathbf{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} - C\right)\right] = \mathbf{g}_L(\boldsymbol{\theta}) + \mu\mathbf{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} = \mathbf{0} \tag{6.23}$$

Solving this equation we get the optimal increment:

$$\Delta\boldsymbol{\theta}^* = -\frac{1}{\mu}\mathbf{F}^{-1}(\boldsymbol{\theta})\mathbf{g}_L(\boldsymbol{\theta}) \tag{6.24}$$

and the optimal parameter can be found iteratively using this method of *natural gradient descent* (Eq. (B.3.5):

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \Delta\boldsymbol{\theta}_n = \boldsymbol{\theta}_n + \lambda\mathbf{F}^{-1}\mathbf{g}_L(\boldsymbol{\theta}_n) = \boldsymbol{\theta}_n + \lambda\tilde{\mathbf{g}}_L(\boldsymbol{\theta}_n) \tag{6.25}$$

where $\lambda = 1/\mu$ is the step size, and $\tilde{\mathbf{g}}_L(\boldsymbol{\theta}_n)$ is the *natural gradient*:

$$\tilde{\mathbf{g}}_L(\boldsymbol{\theta}_n) = \mathbf{F}^{-1}\mathbf{g}_L(\boldsymbol{\theta}_n) \tag{6.26}$$

Comparing this iteration with that in Eq. (2.15), we see that the natural gradient method is similar to the Newton-Raphson method in the sense that it is based on not only the gradient $\mathbf{g}_J(\boldsymbol{\theta})$ for the local first order variation of the likelihood function $L(\boldsymbol{\theta})$ in the distribution space, but also the Fisher information $\mathbf{F}(\boldsymbol{\theta})$ for the more global second order curvature in the space, by which the gradient direction is modified so that the iterative search process in the space can be carried out more efficiently.

Problems

1. Develop Matlab code to implement the nonlinear least-squares regression algorithm considered in Section 6, and apply your code to resolve the 1-D nonlinear regression problem in Example 6.1 The datasets can be downloaded from here....
2. Apply your code to resolve the 2-D nonlinear regression problem such as that in Example 6.2. Three data sets can be found below to find the optimal model parameters (a, b, c, d) for your models.
   - dataset 1 in /e176/homework_e190/data3.txt
   - dataset 2 in /e176/homework_e190/data4.txt
   - dataset 3 in /e176/homework_e190/data5.txt

   You can read these data into your Matlab code by `data=load('data3.txt')`. Each data file is organized in three columns, the first two are for the two independent variables x_1 and x_2 representing a set of data points in a 2-D space, while the third one is for the function values $f(x_1, x_2)$ corresponding to these points.

7 Logistic and Softmax Regression

7.1 Logistic Regression and Binary Classification

As mentioned previously, regression methods can be used as a binary classifier, when the regression function $f(\mathbf{x}, \boldsymbol{\theta})$ is thresholded by some constant c, which is assumed to be zero in the following without loss of generality. Once the model parameter $\boldsymbol{\theta}$ is obtained based on the training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$, every point $\mathbf{x}$ in the d-dimensional feature space can be classified into either of two classes C_- and C_+:

$$\text{if } f(\mathbf{x}) \begin{cases} < c \\ > c \end{cases} \quad \text{then} \quad \mathbf{x} \in \begin{cases} C_- \\ C_+ \end{cases} \tag{7.1}$$

When $c = 0$, the regression function $f(\mathbf{x})$ is thresholded by a unit step function $u(x)$ ($u(x) = 1$ if $x \geq 0$ but 0 otherwise), and the corresponding equation $f(\mathbf{x}) = 0$ represents a hypersurface in the d-dimensional space (a point if $d = 1$, a curve if $d = 2$, a surface or hypersurface if $d \geq 3$), by which the space is partitioned into two regions. In this case, $f(\mathbf{x})$ is called the *decision function* and the surface *decision boundary*. Now any point in the space is classified into either of the two classes, depending on whether it is on the positive side of the decision boundary ($f(\mathbf{x}) > 0$) or the negative side ($f(\mathbf{x}) < 0$).

This simple binary classifier suffers from the drawback that the classification result is binary and deterministic. It provides no additional information such as how reliable the result is, i.e., how confident we can be regarding the result. Specifically, such a classifier simply classifies all points on one side of the boundary into the same class, without considering how far they are from the decision boundary, while one would naturally feel more confident about the classification of points far away from the boundary, than those close to it. In other words, this binary classifier may tend to overfit as it is sensitive to noisy data points close to the boundary.

This problem can be addressed by the method of *logistic regression*, which can provide the probability for any given data point to be classified to either of the two classes, as a quantitative measurement of the confidence for the classification. Based on such a probabilistic result, one can not only make a binary decision, but also gain the extra information in terms of the reliability of the decision.

The logistic regression is based on the same linear regression function $f(\mathbf{x}) =$

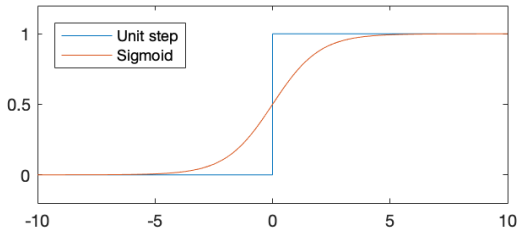


Figure 7.1 Unit Step vs. Sigmoid

$\mathbf{w}^T\mathbf{x}$. However, different from linear regression, in logistic regression, the linear function is not hard-thresholded by the unit step function to become either 0 or 1; instead, it is soft-thresholded by a sigmoid function $\sigma(f(\mathbf{x})) = \sigma(\mathbf{w}^T\mathbf{x})$ to produce a value between 0 and 1, as shown in Fig. 7.1, which can then be interpreted as the probability $p(\mathbf{x} \in C_+)$ for $\mathbf{x}$ to belong to C_+, and the probability for $\mathbf{x}$ to belong to C_- is simply $p(\mathbf{x} \in C_-) = 1 - p(\mathbf{x} \in C_+)$.

Similar to linear and nonlinear regression methods considered before, the goal of logistic regression is also to find the optimal model parameter $\mathbf{w}$ for the regression function $f(\mathbf{x}, \mathbf{w})$ to fit the training set optimally. But different from linear and nonlinear regression based on the least squares (LS) method that minimizes the squared error in Eq. (6.3), logistic regression is based on maximum likelihood (ML) estimation, which maximizes the *likelihood* of the model parameter, as discussed below.

The sigmoid function can be either the error function (cumulative Gaussian, erf):

$$\phi(x) = \int_{-\infty}^{x} \mathcal{N}(u|0,1)\, du = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{1}{2N}u^2\right) du \tag{7.2}$$

or the logistic function:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} = \begin{cases} 1 & x \to \infty \\ 1/2 & x = 0 \\ 0 & x \to -\infty \end{cases} \tag{7.3}$$

In the following, we will only consider the logistic function for consistency and simplicity. Here is a set of properties of the logistic function:

$$\begin{aligned} \sigma(-x) &= \frac{1}{1+e^x} = 1 - \frac{1}{1+e^{-x}} = 1 - \sigma(x) \\ \frac{d}{dx}\sigma(x) &= \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)\sigma(-x) = \sigma(x)(1-\sigma(x)) \\ x &= \ln\left(\frac{\sigma(x)}{1-\sigma(x)}\right), \qquad e^x = \frac{\sigma(x)}{1-\sigma(x)} \end{aligned} \tag{7.4}$$

Moreover, a parameter w can be included in the logistic function to control the softness or smoothness of the transition between 0 and 1, as shown in Fig. 7.2.

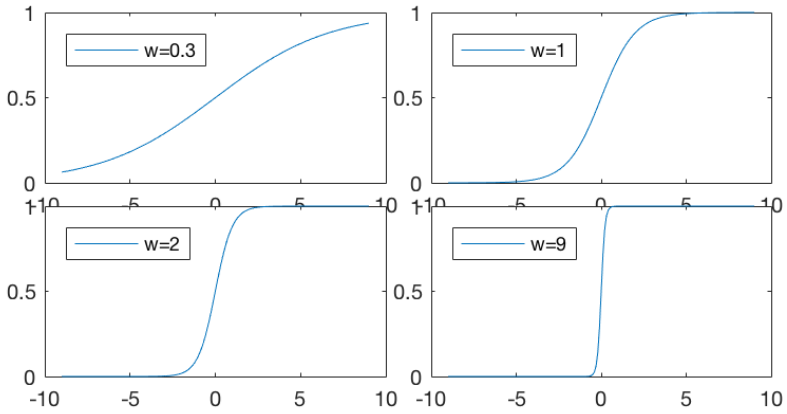


Figure 7.2 Sigmoid with Different Parameter w

$$\sigma(wx) = \begin{cases} 1/2 & w = 0 \\ 1/(1+e^{-wx}) & w > 0 \\ u(x) & w \to \infty \end{cases} \tag{7.5}$$

We see that the argument x is thresholded by the logistic function with variable softness controlled by parameter w, between the two extreme cases of $\sigma(w\,x) = 1/2$ (no thresholding) when $w = 0$, and $\sigma(w\,x) = u(x)$ when $w \to \infty$ (hard thresholding).

Now the linear function $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \in (-\infty,\ \infty)$ is mapped by the sigmoid function $\sigma(f(\mathbf{x}))$ to a real value between 0 and 1, which is interpreted as the conditional probability for output $\hat{y} = 1$ given data point $\mathbf{x}$ and model parameter $\mathbf{w}$:

$$p(\hat{y} = 1|\mathbf{x}, \mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}} \in (0,\ 1) \tag{7.6}$$

Comparing this with Eq. (7.5) for $w > 0$, we note that the weight vector $\mathbf{w}$ controls the smoothness of the thresholoding.

This probability $p(\hat{y} = 1|\mathbf{x}, \mathbf{w})$ is also the probability $p(\mathbf{x} \in C_+)$ for $\mathbf{x} \in C_+$, and the probability $p(\mathbf{x} \in C_-)$ is:

$$p(\hat{y} = 0|\mathbf{x}, \mathbf{w}) = 1 - p(y = 1|\mathbf{x}, \mathbf{w}) = 1 - \sigma(\mathbf{w}^T\mathbf{x}) = \sigma(-\mathbf{w}^T\mathbf{x}) = \frac{1}{1+e^{\mathbf{w}^T\mathbf{x}}} \tag{7.7}$$

Now any given $\mathbf{x}$ can be classified into either class C_+ or class C_- depending on if $\sigma(\mathbf{w}^T\mathbf{x}) > 1/2$. Specifically, consider the following cases:

- If $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \gg 0$, i.e., $\mathbf{x}$ is on the positive side far away from the plane $f(\mathbf{x}) = 0$, then the probability $p(\mathbf{x} \in C_+) = \sigma(f(\mathbf{x}))$ is close to 1;
- If $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \ll 0$, i.e., $\mathbf{x}$ is on the negative side far away from the plane

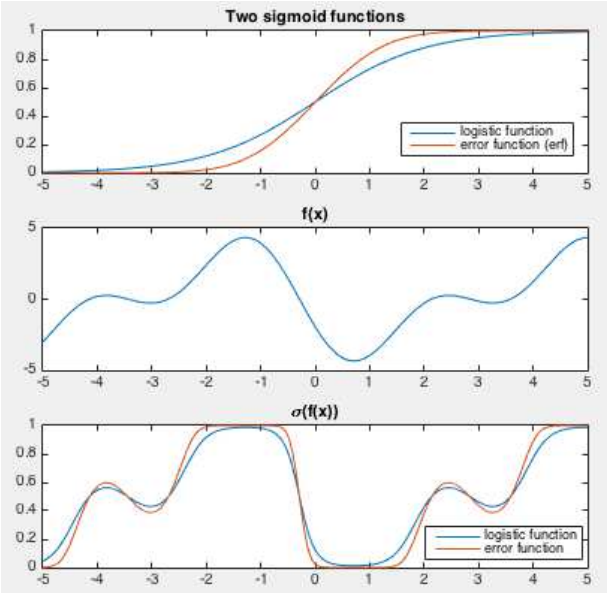


Figure 7.3 Convert Regression Function to Probability

$f(\mathbf{x}) = 0$, then the probability $p(\mathbf{x} \in C_+) = \sigma(f(\mathbf{x}))$ is close to 0, and $p(\mathbf{x} \in C_-) = 1 - p(\mathbf{x} \in C_+)$ is close to 1;

- If $f(\mathbf{x}) \approx 0$, then both $p(\mathbf{x} \in C_+) = \sigma(f(\mathbf{x}))$ and $p(\mathbf{x} \in C_-) = 1 - \sigma(f(\mathbf{x}))$ are approximately 1/2, i.e., if a point $\mathbf{x}$ is very close to the plane $f(\mathbf{x}) = 0$, then the probabilities for it to belong to either C_+ or C_- are about the same.

We note that the weight vector $\mathbf{w}$ plays the same role as w in Eq. (7.5) ($w > 0$), i.e., a larger value of $||\mathbf{w}||$ results in a sharper thresholding by the sigmoid function $\sigma(\mathbf{w}^T\mathbf{x})$. In other words, $\mathbf{w}$ determines not only how the classification is carried out based on the decision boundary, but also whether the classifier over and underfits, as further illustrated in the two extreme cases:

- $||\mathbf{w}|| \to \infty$: the value of the sigmoid function $\sigma(\mathbf{w}^T\mathbf{x})$ is approximately binary, close to either 0 or 1, based on which the feature space is hard-divided into two regions each for one of the two classes. As such a binary classification is dictated mostly by a small number of data points close to the boundary between the two regions, it is prone to noise contained in these data points and tends to overfit.
- $||\mathbf{w}|| \to 0$: the sigmoid function approaches a constant $\sigma(\mathbf{w}^T\mathbf{x}) = 1/2$, totally insensitive to the locations of the data points, i.e., the probability for any data point to belong to either of the two classes is always the same. As such a classifier is insensitive to the valid variations in the data, it tends to underfit.

We therefore see that by adjusting $||\mathbf{w}||$ and thereby controlling the smoothness of the thresholding, we can make a proper tradeoff between overfitting and underfitting.

We label the each sample $\mathbf{x}_n$ in the training set $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ by either $y_n = 1$ if $\mathbf{x}_n \in C_+$, or $y_n = 0$ if $\mathbf{x}_n \in C_-$. Then the two probabilities in Eqs. (7.6) and (7.7) can be combined into the following conditional probability for the model output $\hat{y}$ to match the ground truth labeling y_n:

$$\begin{aligned} p(\hat{y} = y_n|\mathbf{x}_n, \mathbf{w}) &= P(\hat{y} = 1|\mathbf{x}_n, \mathbf{w})^{y_n} p(\hat{y} = 0|\mathbf{x}_n, \mathbf{w})^{1-y_n} \\ &= \begin{cases} p(\hat{y} = 1|\mathbf{x}_n, \mathbf{w}) & \text{if } y_n = 1 \\ p(\hat{y} = 0|\mathbf{x}_n, \mathbf{w}) & \text{if } y_n = 0 \end{cases} \end{aligned} \tag{7.8}$$

We can further get the likelihood function of $\mathbf{w}$ given all N samples in $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ (assumed i.i.d.):

$$\begin{aligned} L(\mathbf{w}|\mathcal{D}) \propto p(\mathcal{D}|\mathbf{w}) = p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} p(\hat{y} = y_n|\mathbf{x}_n, \mathbf{w}) \\ = \prod_{n=1}^{N} p(\hat{y} = 1|\mathbf{x}_n)^{y_n} p(\hat{y} = 0|\mathbf{x}_n)^{1-y_n} = \prod_{n=1}^{N} \sigma(\mathbf{w}^T\mathbf{x}_n)^{y_n} (1 - \sigma(\mathbf{w}^T\mathbf{x}_n))^{1-y_n} \end{aligned} \tag{7.9}$$

The optimal model parameter $\mathbf{w}$ can be found by either of the two methods below:

- *Maximum likelihood estimateion (MLE)*

  The optimal model parameter $\mathbf{w}$ is defined as the one that maximizes the likelihood function $L(\mathbf{w}|\mathcal{D})$, or, equivalently, that minimizes the negative log likelihood as the objective function:

  $$\begin{aligned} J(\mathbf{w}) &= -l(\mathbf{w}|\mathcal{D}) = -\log L(\mathbf{w}|\mathcal{D}) \\ &= -\sum_{n=1}^{N} \left[y_n \log \sigma(\mathbf{w}^T\mathbf{x}_n) + (1 - y_n) \log(1 - \sigma(\mathbf{w}^T\mathbf{x}_n)) \right] \end{aligned} \tag{7.10}$$

  This is the general MLE method that is widely used to obtain the most probable estimation of certain parameter of a model based on the given data.

- *Maximum a posteriori (MAP)*

  The optimal parameter $\mathbf{w}$ can also be defined as the one that maximizes the posterior probability:

  $$p(\mathbf{w}|\mathcal{D}) = p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{X}, \mathbf{w})\ p(\mathbf{w})}{p(\mathbf{y}|\mathbf{X})} \propto p(\mathbf{y}|\mathbf{X}, \mathbf{w})\ p(\mathbf{w}) \tag{7.11}$$

  where $p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = L(\mathbf{w}|\mathcal{D})$ is the likelihood given in Eq. (7.9), $p(\mathbf{w})$ is the prior probability of $\mathbf{w}$ before observing the training data $\mathcal{D}$, and the denominator $p(\mathbf{yX})$ is dropped as it is independent of variable $\mathbf{w}$.

  Based on the assumption that all components of $\mathbf{w}$ are independent of

each other and they have zero mean and the same variance, we let the prior probability be:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I}) = \frac{1}{(2\pi\sigma^2)^{d/2}} \exp\left(-\frac{||\mathbf{w}||^2}{2\sigma^2}\right) \propto \exp\left(-\frac{||\mathbf{w}||^2}{2\sigma^2}\right) \quad (7.12)$$

where the normalizing factor $1/(2\pi\sigma^2)^{d/2}$ independent of $\mathbf{w}$ can be dropped. The assumed zero mean is due to the desire that the values of all weights in $\mathbf{w}$ should be around zero instead of large values on either positive or negative side, to avoid overfitting, as discussed below.

Now the posterior can be written as

$$\begin{aligned} p(\mathbf{w}|\mathcal{D}) \;&\propto\; p(\mathbf{y}|\mathbf{X}, \mathbf{w})\; p(\mathbf{w}) = \prod_{n=1}^{N} p(\hat{y}_n = y_n|\mathbf{x}_n, \mathbf{w})\; \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_w) \\ &\propto\; \prod_{n=1}^{N} \sigma(\mathbf{w}^T\mathbf{x}_n)^{y_n}(1 - \sigma(\mathbf{w}^T\mathbf{x}_n))^{1-y_n}\; \exp\left(-\frac{||\mathbf{w}||^2}{2\sigma^2}\right) \quad (7.13) \end{aligned}$$

The MAP method finds the optimal model parameter $\mathbf{w}$ that maximizes this posterior $p(\mathbf{w}|\mathcal{D})$ or equivalently minimizes the negative log posterior as the objective function:

$$\begin{aligned} J(\mathbf{w}) &= -\log p(\mathbf{w}|\mathcal{D}) = -\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) - \left(-\frac{||\mathbf{w}||^2}{2\sigma^2}\right) \\ &= -\sum_{n=1}^{N} \left[y_n \log \sigma(\mathbf{w}^T\mathbf{x}) + (1 - y_n)\log(1 - \sigma(\mathbf{w}^T\mathbf{x}))\right] + \frac{\lambda}{2}||\mathbf{w}||^2 \quad (7.14) \end{aligned}$$

where λ is a hyperparameter defined as $\lambda = 1/\sigma^2$. Comparing this expression with Eq. (5.32) for ridge regression, we note that the second term can be treated as a penalty term that discourages large values of $\mathbf{w}$ to avoid overfitting. When $J(\mathbf{w})$ is minimized, a larger value of λ results in smaller values for $\mathbf{w}$, and the sigmoid function will be more smooth for a softer thresholding, and the classification is less affected by the possible noisy outliers close to the decision boundary. By adjusting λ, we can make a proper tradeoff between overfitting and underfitting.

In particular, if $\sigma \to \infty$ and $\lambda \to 0$, then $\exp(-\lambda||\mathbf{w}||^2/2) \to 1$, i.e., the prior $p(\mathbf{w})$ approaches a uniform distribution (all values of $\mathbf{w}$ are equally likely), then the penalty term can be removed, and Eq. (7.14) becomes the same as Eq. (7.10), i.e., the MAP solution that maximizes the posterior $p(\mathbf{w}|\mathcal{D})$ becomes the same as the MLE solution that maximizes the likelihood $L(\mathbf{w}|\mathcal{D})$.

Here we consider the MAP method which contains an extra term $||\mathbf{w}||^2//2\sigma^2)$

and is therefore more general. We first find the gradient of objective function:

$$\begin{aligned}
\mathbf{g}_J(\mathbf{w}) &= \frac{d}{d\mathbf{w}} J(\mathbf{w}) = \frac{d}{d\mathbf{w}} \left(-\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) + \frac{\lambda}{2}||\mathbf{w}||^2 \right) \\
&= -\sum_{n=1}^{N} \frac{d}{d\mathbf{w}} \left[y_n \log \sigma(\mathbf{w}^T \mathbf{x}_n) + (1 - y_n) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_n)) \right] + \lambda \mathbf{w} \\
&= -\sum_{n=1}^{N} \left[\frac{y_n}{\sigma(\mathbf{w}^T \mathbf{x}_n)} - \frac{1 - y_n}{1 - \sigma(\mathbf{w}^T \mathbf{x}_n)} \right] \sigma(\mathbf{w}^T \mathbf{x}_n)(1 - \sigma(\mathbf{w}^T \mathbf{x}_n))\, \mathbf{x}_n + \lambda \mathbf{w} \\
&= \sum_{n=1}^{N} \left[\sigma(\mathbf{w}^T \mathbf{x}_n) - y_n \right] \mathbf{x}_n + \lambda \mathbf{w} = [\mathbf{x}_1, \cdots, \mathbf{x}_N] \begin{bmatrix} \sigma(\mathbf{w}^T \mathbf{x}_1) - y_1 \\ \vdots \\ \sigma(\mathbf{w}^T \mathbf{x}_N) - y_N \end{bmatrix} + \lambda \mathbf{w} \\
&= \mathbf{X}\mathbf{r} + \lambda \mathbf{w}
\end{aligned} \tag{7.15}$$

where we have used the properties of the logistic function in Eq. (7.4), and $\mathbf{r}(\mathbf{w})$ is the residual defined as

$$\mathbf{r} = \mathbf{r}(\mathbf{w}) = \begin{bmatrix} r_1(\mathbf{w}) \\ \vdots \\ r_N(\mathbf{w}) \end{bmatrix} = \begin{bmatrix} \sigma(\mathbf{w}^T \mathbf{x}_1) - y_1 \\ \vdots \\ \sigma(\mathbf{w}^T \mathbf{x}_N) - y_N \end{bmatrix} \tag{7.16}$$

containing all N residuals $r_n = \sigma(\mathbf{w}^T \mathbf{x}_n) - y_n \;\; (n = 1, \cdots, N)$, the difference between the model prediction $0 < \sigma(\mathbf{w}^T \mathbf{x}_n) < 1$ and the ground truth labeling $y_n \in \{0, 1\}$.

Based on $\mathbf{g}_J(\mathbf{w})$ given in Eq. (7.15), we can find the optimal $\mathbf{w}$ that minimizes this objective function by the either the gradient descent method or Newton's method that depends on the second order Hessian matrix $\mathbf{H}_J(\mathbf{w})$ in addition to the first order gradient.

The Matlab code segment below is for the gradient descent method. Here `X` contains the N samples $\mathbf{x}_1, \cdots, \mathbf{x}_N$ nd `y` contains the corresponding a binary labeling:

```
X=[ones(1,n); X];       % augmented X with x_0=1
w=ones(m+1,1);          % initialize weight vector
g=X*(sgm(w,X)-y)';      % gradient of log posterior
tol=10^(-6);            % tolerance
delta=0.1;              % step size
while norm(g)>tol       % terminate when g is small enough
    w=w-delta*g;        % update weight by gradient descent
    g=X*(sgm(w,X)-y)'; % update gradient
end
```

where

```
function s=sgm(w,X)
    s=1./(1+exp(-w'*X));
```

```
end
```

We can also use Newton's method if the second order Hessian matrix is available:

$$\begin{aligned}\mathbf{H}_J(\mathbf{w}) &= \frac{d}{d\mathbf{w}}\mathbf{g}_J(\mathbf{w}) = \frac{d}{d\mathbf{w}}(\mathbf{X}\mathbf{r}(\mathbf{w}) + \lambda||\mathbf{w}||) = \mathbf{X}\frac{d}{d\mathbf{w}}\mathbf{r}(\mathbf{w}) + \lambda \\ &= \mathbf{X}\frac{d}{d\mathbf{w}}\begin{bmatrix} r_1(\mathbf{w}) \\ \vdots \\ r_N(\mathbf{w}) \end{bmatrix} + \lambda = \mathbf{X}\mathbf{J}_r(\mathbf{w}) + \lambda \end{aligned} \tag{7.17}$$

where $\mathbf{J}_r(\mathbf{w}) = d\mathbf{r}(\mathbf{w})/d\mathbf{w}$ is the $N \times (d+1)$ Jacobian matrix of the vector function $\mathbf{r}(\mathbf{w})$, of which the nth row is

$$\frac{dr_n(\mathbf{w})}{d\mathbf{w}} = \frac{d}{d\mathbf{w}}\left[\sigma(\mathbf{w}^T\mathbf{x}_n) - y_n\right] = \sigma(\mathbf{w}^T\mathbf{x}_n)[1 - \sigma(\mathbf{w}^T\mathbf{x}_n)]\mathbf{x}_n = z_n\mathbf{x}_n \tag{7.18}$$

where we have defined $z_n = \sigma(\mathbf{w}^T\mathbf{x}_n)(1 - \sigma(\mathbf{w}^T\mathbf{x}_n))$, which is always greater than zero as $0 < \sigma(\mathbf{w}^T\mathbf{x}_n) < 1$. Now the Jacobian can be written as

$$\mathbf{J}_r(\mathbf{w}) = \begin{bmatrix} dr_1(\mathbf{w})/d\mathbf{w} \\ \vdots \\ dr_N(\mathbf{w})/d\mathbf{w} \end{bmatrix} = \begin{bmatrix} z_1\mathbf{x}_1 \\ \vdots \\ z_N\mathbf{x}_N \end{bmatrix} = \begin{bmatrix} z_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & z_N \end{bmatrix}\begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} = \mathbf{Z}\mathbf{X}^T \tag{7.19}$$

where $\mathbf{Z} = \text{diag}(z_1, \cdots, z_N)$ is a positive definite diagonal matrix (as $z_n > 0$). Substituting $\mathbf{J}_r$ back into Eq. (7.17), we get

$$\mathbf{H}_J = \mathbf{X}\mathbf{J}_r + \lambda = \mathbf{X}\mathbf{Z}\mathbf{X}^T + \lambda \tag{7.20}$$

Since $\mathbf{Z}$ is positive definite, the following quadratic function is also positive, i.e., for any $\mathbf{y}$, we have

$$\mathbf{y}^T\mathbf{H}_J\mathbf{y} = \mathbf{y}^T\mathbf{X}\mathbf{Z}\mathbf{X}^T\mathbf{y} + \lambda\mathbf{y}^T\mathbf{y} = (\mathbf{X}^T\mathbf{y})^T\mathbf{Z}(\mathbf{X}^T\mathbf{y}) + \lambda||\mathbf{y}|| > 0 \tag{7.21}$$

indicating $\mathbf{H}_J$ is also positive definite, and the objective function $J(\mathbf{w})$ when approximated by the first three terms of its Taylor series is has a global minimum.

Having found $\mathbf{H}_J(\mathbf{w})$ as well as $\mathbf{g}_l = J(\mathbf{w})$ of the objective function $J(\mathbf{w})$, we can solve the minimization problem also by Newton's method:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \mathbf{H}_J(\mathbf{w}_n)^{-1}\mathbf{g}_J(\mathbf{w}_n) = \mathbf{w}_n - (\mathbf{X}\mathbf{Z}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{r} \tag{7.22}$$

The following code segment estimates weight vector $\mathbf{w}$ by Newton method:

```
X=[ones(1,n); X];          % augmented X with x_0=1
w=ones(m+1,1);             % initialize weight vector
g=X*(sgm(w,X)-y)';         % gradient of log posterior
H=X*diag(s.*(1-s))*X';     % Hessian of log posterior
tol=10^(-6)
while norm(g)>tol          % terminate when g is small enough
    w=w-inv(H)*g;          % update weight by Newton-Raphson
```

```
    s=sgm(w,X);             % update sigma
    g=X*(s-y)';             % update gradient
    H=X*diag(s.*(1-s))*X'; % update Hessian
end
```

Once the model parameter $\mathbf{w}$ is available, an unlabeled test sample $\mathbf{x}_*$ of unknown class can be classified into either of the two classes C_1 and C_0, based on $p(y_*=1|\mathbf{x}_*)$:

$$\text{if} \quad p(y_*=1|\mathbf{x}_*)=\sigma(\mathbf{w}^T\mathbf{x}_*)\begin{cases} >1/2 \\ <1/2 \end{cases}, \quad \text{then} \quad \mathbf{x}_* \in \begin{cases} C_1 \\ C_0 \end{cases} \tag{7.23}$$

In summary, we make a comparison between the linear and logistic regression methods, both can be used for binary classification. In linear regression, the function value $f(\mathbf{x})=\mathbf{w}^T\mathbf{x}$ is hard-thresholded by the unit step function, so that any data point $\mathbf{x}$ is classified to class C_+ if $\mathbf{w}^T\mathbf{x}>0$, or class C_- if $\mathbf{w}^T\mathbf{x}<0$ in a deterministic fashion (with probability $p(\mathbf{x}\in C_+)\in\{0,\,1\}$. On the other hand, in logistic regression, the linear function $f(\mathbf{x})=\mathbf{w}^T\mathbf{x}$ is soft-thresholded by the sigmoid function to become $p(\mathbf{x}\in C_+)=\sigma(f(\mathbf{x}))\in(0,\;1)$, treated as the probability for $\mathbf{x}$ to belong to class C_+. Although the same linear model $f(\mathbf{x})=\mathbf{w}^T\mathbf{x}$ is used in both cases, the two methods obtain the model parameter $\mathbf{w}$ by minimizing very different objective functions (Eqs. (5.8) and (7.14)).

7.2 Softmax Regression for Multiclass Classification

In a multiclass classification problem, an unlabeled data point $\mathbf{x}$ is to be classified into one of $K>2$ classes $\{C_1,\cdots,C_K\}$, based on the training set $\mathcal{D}=\{(\mathbf{x}_n,y_n),\;\; n=1,\cdots,N\}$, where $y_n\in\{1,\cdots,K\}$ is an integer indicating $\mathbf{x}_n\in C_{y_n}$

Any binary classifier, such as the logistic regression considered above, can be used to solve such a multiclass classification problem in either of the following two ways:

- One-Vs-One (1V1): construct $K(K-1)/2$ binary classifiers for every pair of the K classes based on the training set $\{(C_i,\,C_j)\quad i=2,\cdots,K,\;\; j=1,\cdots,i-1\}$, then an unlabeled sample $\mathbf{x}$ is classified to one of the K classes with maximum votes out of this many binary classifications.
- One-Vs-Rest (1VR). Convert a K-class problem into K binary problems by regrouping the K classes into two classes $C_+=C_i$ and $C_-=\{C_j|j=1,\cdots,K,\;\; j\neq i\}$, and get the corresponding discriminant function $f(\mathbf{x})$ of the binary problem:

$$\text{if } f_i(\mathbf{x})\begin{cases} >0 \\ <0 \end{cases}, \text{ then } \begin{cases} \mathbf{x}\in C_+=C_i \\ \mathbf{x}\in C_- \end{cases} \tag{7.24}$$

  which represents quantitatively how much a given $\mathbf{x}$ belongs to class $C_+=$

C_i, instead of C_- containing the rest $K-1$ classes. This process is repeated K times for all $i = 1, \cdots, K$ to get

$$f_k(\mathbf{x}) = \max\{f_1(\mathbf{x}), \cdots, f_K(\mathbf{x})\}, \qquad (k = 1, \cdots, K) \tag{7.25}$$

which indicates $\mathbf{x}$ belongs to class C_k.

Alternatively, a multiclass problem with $K > 2$ can also be solved by *multinomial logistic* or *softmax regression*, which can be considered as a generalized version of the logistic regression method, based on the *softmax function* of K variables $z_1, \cdots, z_K$:

$$s_k = s(z_k) = \frac{e^{z_k}}{\sum_{l=1}^{K} e^{z_l}}, \qquad (k = 1, \cdots, K) \tag{7.26}$$

with the following properties:

$$0 < s_k < 1, \qquad \sum_{k=1}^{K} s_k = 1 \tag{7.27}$$

and

$$\begin{aligned} \frac{\partial s(z_i)}{\partial z_j} &= \frac{\delta_{ij} e^{z_i} \sum_{l=1}^{K} e^{z_l} - e^{z_i} e^{z_j}}{\left(\sum_{l=1}^{K} e^{z_l}\right)^2} \\ &= \delta_{ij} s(z_i) - s(z_i) s(z_j) = s(z_i)[\delta_{ij} - s(z_j)] \end{aligned} \tag{7.28}$$

where δ_{ij} is the Dirac delta function which is 1 if $i = j$, but 0 otherwise.

Similar to the sigmoid function used to model the conditional probability $p(\hat{y} = 1|\mathbf{x}, \mathbf{w})$ for any data point $\mathbf{x}$ to belong to class C_+ based on model parameter $\mathbf{w}$ in Eq. (7.6), here the softmax function s_k defined above is used to model the conditional probability for $\mathbf{x}$ to belong to class C_k based on some model parameter $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_K]$:

$$p(\hat{y} = k|\mathbf{x}, \mathbf{W}) = s_k = s(\mathbf{w}_k^T \mathbf{x}) = \frac{e^{\mathbf{w}_k^T \mathbf{x}}}{\sum_{l=1}^{K} e^{\mathbf{w}_l^T \mathbf{x}}}, \qquad (k = 1, \cdots, K) \tag{7.29}$$

where $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_K]$ is composed of K weight vectors each associated with one of the K classes $\{C_1, \cdots, C_K\}$, to be determined in the training process based on training set $\mathcal{D}$. Same as in logistic regression, here both $\mathbf{x}$ and $\mathbf{w}$ are augmented $d + 1$ dimensional vectors.

We note that the inner product $\mathbf{w}^T \mathbf{x} = ||\mathbf{w}|| \, ||\mathbf{x}|| \cos\theta$ of vectors $\mathbf{w}$ and $\mathbf{x}$ is inversely related to the angle θ between the two vectors, i.e., when compared with other classes, a data sample $\mathbf{x}$ has a smaller angular distance to $\mathbf{w}_k$, then it has a larger inner product $\mathbf{w}_k^T \mathbf{x}$ and thereby greater probability $p(\hat{y} = k|\mathbf{x})$ to belong to class C_k. We therefore see that weight vectors $\mathbf{w}_1, \cdots, \mathbf{w}_K$ as the parameters of the softmax regression model actually represent the angular directions of the corresponding classes in the feature space.

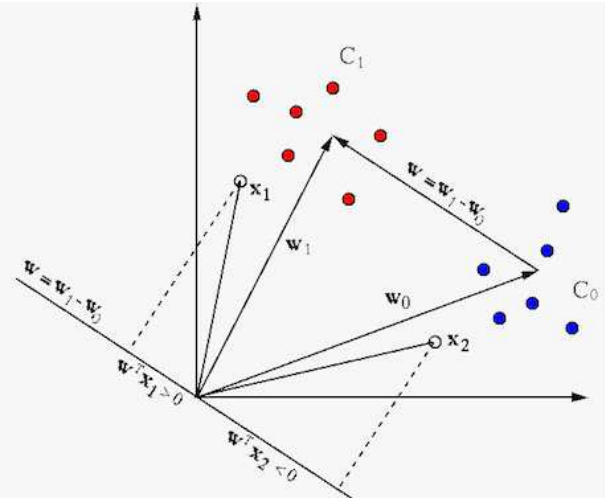


Figure 7.4 Softmax Classification

Specially, when $K = 2$, the softmax function defined in Eq. (7.29) becomes the logistic function:

$$s_1 = \frac{e^{\mathbf{w}_1^T\mathbf{x}}}{e^{\mathbf{w}_0^T\mathbf{x}} + e^{\mathbf{w}_1^T\mathbf{x}}} = \frac{1}{1 + e^{-(\mathbf{w}_1 - \mathbf{w}_0)^T\mathbf{x}}} = \sigma(\mathbf{w}^T\mathbf{x})$$
$$s_0 = \frac{e^{\mathbf{w}_0^T\mathbf{x}}}{e^{\mathbf{w}_0^T\mathbf{x}} + e^{\mathbf{w}_1^T\mathbf{x}}} = \frac{1}{1 + e^{(\mathbf{w}_1 - \mathbf{w}_0)^T\mathbf{x}}} = \sigma(-\mathbf{w}^T\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x}) \quad (7.30)$$

where we have defined $\mathbf{w} = \mathbf{w}_1 - \mathbf{w}_0$, as the vector between the two classes of C_0 and C_1 (or C_+ and C_-), which is the normal direction of a decision plane separating the two classes, as shown in Fig. 7.4 for a 2-D case. Any point $\mathbf{x}$ of unknown class can be therefore classified into either of the two classes depending on whether its projection $\mathbf{w}^T\mathbf{x}$ onto $\mathbf{w}$ is positive or negative, i.e., on which side of the decision plane it is located.

For mathematical convenience, we also label each sample $\mathbf{x}_n$ in the training set by a binary vector $\mathbf{y}_n = [y_{n1}, \cdots, y_{nK}]^T$, in addition to its integer labeling $y_n \in \{1, \cdots, K\}$. If $y_n = k$ indicating $\mathbf{x}_n \in C_k$, then the kth component of $\mathbf{y}_n$ is $y_{nk} = 1$, while other components are zero $y_{nl} = 0$ for all $l \neq k$. Note that all components of $\mathbf{y}_n$ add up to 1. The N training samples in $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ are also labeled by their corresponding binary labelings in $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$.

We define the probability for $\mathbf{x}_n \in C_k$ as the softmax function based on the linear function $\mathbf{w}_k^T\mathbf{x}_n$:

$$s_{nk} = p(\hat{y} = k|\mathbf{x}_n) = \frac{e^{\mathbf{w}_k^T\mathbf{x}_n}}{\sum_{l=1}^K e^{\mathbf{w}_l^T\mathbf{x}_n}}, \qquad (k = 1, \cdots, K, \;\; n = 1, \cdots, N) \quad (7.31)$$

and the probability for $\mathbf{x}_n$ to be correctly classified into class C_{y_n} can be written

as the following product of K factors (of which $K-1$ are equal to 1):

$$p(\hat{y}=y_n|\mathbf{x}_n,\mathbf{W})=\prod_{k=1}^{K}p(\hat{y}=k|\mathbf{x}_n,\mathbf{W})^{y_{nk}}=\prod_{k=1}^{K}s_{nk}^{y_{nk}} \tag{7.32}$$

Our goal is to find these weight vectors in $\mathbf{W}=[\mathbf{w}_1,\cdots,\mathbf{w}_K]$ as the parameter of the softmax model for it to optimally fit the N i.i.d. data points in the training set, so that that the following likelihood is maximized:

$$\begin{aligned}L(\mathbf{W}|\mathcal{D})\propto p(\mathcal{D}|\mathbf{W})&=\prod_{n=1}^{N}p(\hat{y}=y_n|\mathbf{x}_n,\mathbf{W})=\prod_{n=1}^{N}\left(\prod_{k=1}^{K}s_{nk}^{y_{nk}}\right)\\&=\prod_{n=1}^{N}\prod_{k=1}^{K}\left(\frac{e^{\mathbf{w}_k^T\mathbf{x}_n}}{\sum_{l=1}^{K}e^{\mathbf{w}_l^T\mathbf{x}_n}}\right)^{y_{nk}}\end{aligned} \tag{7.33}$$

Equivalently we can also minimize the negative log likelihood as the objective function:

$$\begin{aligned}J(\mathbf{W})=-l(\mathbf{W}|\mathcal{D})=-\log L(\mathbf{w}|\mathcal{D})&=-\sum_{n=1}^{N}\sum_{k=1}^{K}y_{nk}\log\left(\frac{e^{\mathbf{w}_k^T\mathbf{x}_n}}{\sum_{l=1}^{K}e^{\mathbf{w}_l^T\mathbf{x}_n}}\right)\\&=-\sum_{n=1}^{N}\sum_{k=1}^{K}y_{nk}\left(\mathbf{w}_k^T\mathbf{x}_n-\log\sum_{l=1}^{K}e^{\mathbf{w}_l^T\mathbf{x}_n}\right)\\&=-\sum_{n=1}^{N}\left[\sum_{k=1}^{K}y_{nk}\mathbf{w}_k^T\mathbf{x}_n-\sum_{k=1}^{K}y_{nk}\left(\log\sum_{l=1}^{K}e^{\mathbf{w}_l^T\mathbf{x}_n}\right)\right]\\&=-\sum_{n=1}^{N}\left[\sum_{k=1}^{K}y_{nk}\mathbf{w}_k^T\mathbf{x}_n-\log\sum_{l=1}^{K}e^{\mathbf{w}_l^T\mathbf{x}_n}\right]\end{aligned} \tag{7.34}$$

The last equality is due to the fact that the last summation of K terms is independent of k and $\sum_{k=1}^{K}y_{nk}=1$. Also, same as in the case of $K=2$, to avoid overfitting, we could further include in this objective function extra regularization terms $\lambda||\mathbf{w}_k||^2/2,\quad(k=1,\cdots,K)$ to penalize large weights for smoother thresholding:

$$J(\mathbf{W})=-l(\mathbf{W}|\mathcal{D})+\frac{\lambda}{2}\sum_{k=1}^{K}||\mathbf{w}_k||^2 \tag{7.35}$$

To find the optimal $\mathbf{W}$ that minimize this objective function $J(\mathbf{W})$, we find

its gradient vector with respective to each of its K columns $\mathbf{w}_i$, $(i = 1, \cdots, K)$:

$$\begin{aligned}
\mathbf{g}_i(\mathbf{W}) &= \frac{d}{d\mathbf{w}_i} J(\mathbf{W}) = \frac{d}{d\mathbf{w}_i} \left[-l(\mathbf{W}|\mathcal{D}) + \frac{\lambda}{2} \sum_{k=1}^{K} ||\mathbf{w}_k||^2 \right] \\
&= -\sum_{n=1}^{N} \left[\sum_{k=1}^{K} y_{nk} \frac{d}{d\mathbf{w}_i} \left(\mathbf{w}_k^T \mathbf{x}_n \right) - \frac{d}{d\mathbf{w}_i} \left(\log \sum_{l=1}^{K} e^{\mathbf{w}_l^T \mathbf{x}_n} \right) \right] + \lambda \mathbf{w}_i \\
&= -\sum_{n=1}^{N} \left[\sum_{k=1}^{K} y_{nk} \delta_{ik} \mathbf{x}_n - \left(\frac{e^{\mathbf{w}_i^T \mathbf{x}_n}}{\sum_{l=1}^{K} e^{\mathbf{w}_l^T \mathbf{x}_n}} \right) \mathbf{x}_n \right] + \lambda \mathbf{w}_i \\
&= \sum_{n=1}^{N} \left(s_{in} - y_{in} \right) \mathbf{x}_n + \lambda \mathbf{w}_i = [\mathbf{x}_1, \cdots, \mathbf{x}_N] \begin{bmatrix} s_{i1} - y_{i1} \\ \vdots \\ s_{iN} - y_{iN} \end{bmatrix} + \lambda \mathbf{w}_i \\
&= [\mathbf{x}_1, \cdots, \mathbf{x}_N] \begin{bmatrix} r_1 \\ \vdots \\ r_N \end{bmatrix} + \lambda \mathbf{w}_i = \mathbf{X} \mathbf{r}_i + \lambda \mathbf{w}_i
\end{aligned} \tag{7.36}$$

where $r_n = s_{in} - y_{in}$ is the residual, the difference between the model output s_{in} and the binary labeling y_{in}, the ith component of $\mathbf{y}_n$ that labels $\mathbf{x}_n$ as well as y_n.

As s_{in} is a function of all K weight vectors in $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_K]$, so is $r_{in} = s_{in} - y_{in}$, and $\mathbf{r}_i$ can be explicitly expressed as $\mathbf{r}_i(\mathbf{W})$. We note that Eq. (7.36) takes the same form as Eq. (7.15) for $K = 2$. When specially $K = 2$ with $k \in \{0,\ 1\}$, the two equations become the same. We therefore see that logistic regression is actually a special case of softmax regression.

We stack all K such $d + 1$ dimensional gradient vectors $\mathbf{g}_1, \cdots, \mathbf{g}_K$ together, and get a $(d + 1)K$ dimensional gradient vector of the objective function $J(\mathbf{W})$ with respect to all K parameter vectors $\mathbf{w}_1, \cdots, \mathbf{w}_K$:

$$\mathbf{g}_J = \begin{bmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_K \end{bmatrix} \tag{7.37}$$

based on which the optimal parameters in $\mathbf{W}$ that minimize the objective function $J(\mathbf{W})$ can be found by the gradient descent method.

We can also find the optimal $\mathbf{W}$ by the Newton's method if we can further find the Hessian matrix $\mathbf{H}_J$ as the second derivative of $J(\mathbf{W})$, by taking the derivative of the ith gradient vector $\mathbf{g}_i(\mathbf{W})$ $(i = 1, \cdots, K)$ with respect to the jth weight vector $\mathbf{w}_j$ $(j = 1, \cdots, K)$ to get the second order derivative of $J(\mathbf{W})$ with respect to both $\mathbf{w}_i$ and $\mathbf{w}_j$, $(i, j = 1, \cdots, K)$:

$$\begin{aligned}
\mathbf{H}_{ij} &= \frac{d^2}{d\mathbf{w}_j d\mathbf{w}_i} J(\mathbf{W}) = \frac{d}{d\mathbf{w}_j} \frac{d}{d\mathbf{w}_i} J(\mathbf{W}) = \frac{d}{d\mathbf{w}_j} \mathbf{g}_i = \frac{d}{d\mathbf{w}_j} \left[\mathbf{X} \mathbf{r}_i + \lambda \mathbf{w}_i \right] \\
&= \mathbf{X} \frac{d\mathbf{r}_i}{d\mathbf{w}_j} + \delta_{ij} \lambda = \mathbf{X} \mathbf{J}_{ij} + \delta_{ij} \lambda
\end{aligned} \tag{7.38}$$

where $\mathbf{J}_{ij} = d\mathbf{r}_i/d\mathbf{w}_j$ is the $N \times (d+1)$ Jacobian matrix of $\mathbf{r}_i$ with respect to $\mathbf{w}_j$, of which the nth row is the transpose of the following vector:

$$\begin{aligned}\frac{d}{d\mathbf{w}_j} r_{in} &= \frac{d}{d\mathbf{w}_j}(s_{in} - y_{in}) = \frac{d}{d\mathbf{w}_j} s_{in} = \frac{d}{d\mathbf{w}_j} s(\mathbf{w}_i^T \mathbf{x}_n) \\ &= s_{in}\left(\delta_{ij} - s_{jn}\right) \frac{d}{d\mathbf{w}_j}\left(\mathbf{w}_j^T \mathbf{x}_n\right) = z_n \mathbf{x}_n \qquad (n = 1, \cdots, N) \end{aligned} \tag{7.39}$$

where we have used Eq. (7.28) and defined $z_n = s_{in}(\delta_{ij} - s_{jn})$. Now the Jacobian can be written as

$$\mathbf{J}_{ij} = \begin{bmatrix} dr_{i1}/d\mathbf{w}_j \\ \vdots \\ dr_{iN}/d\mathbf{w}_j \end{bmatrix} = \begin{bmatrix} z_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & z_N \end{bmatrix} \begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} = \mathbf{Z}\mathbf{X}^T \tag{7.40}$$

where $\mathbf{Z} = \text{diag}(z_1, \cdots, z_N)$ is an $N \times N$ diagonal matrix. Substituting $\mathbf{J}_r(\mathbf{W})$ back into the expression for $\mathbf{H}_{ij}$, we get:

$$\mathbf{H}_{ij} = \mathbf{X}\mathbf{J}_r + \delta_{ij}\lambda = \mathbf{X}\mathbf{Z}\mathbf{X}^T + \delta_{ij}\lambda = \sum_{n=1}^{N} s_{in}(\delta_{ij} - s_{jn})\mathbf{x}_n \mathbf{x}_n^T + \delta_{ij}\lambda \tag{7.41}$$

Here $\mathbf{H}_{ij}$ is a matrix of dimension $(d+1) \times (d+1)$, corresponds to $\mathbf{w}_i$ and $\mathbf{w}_j$ $(i, j = 1, \cdots, K)$. We further stack all $K \times K$ such matrices $\mathbf{H}_{ij}$ together to get the $K(d+1) \times K(d+1)$ dimensional full Hessian matrix $\mathbf{H}_J$ of $J(\mathbf{W})$ with respect to all K vectors in $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_K]$:

$$\mathbf{H}_J = \begin{bmatrix} \mathbf{H}_{11} & \cdots & \mathbf{H}_{1K} \\ \vdots & \ddots & \vdots \\ \mathbf{H}_{K1} & \cdots & \mathbf{H}_{KK} \end{bmatrix} \tag{7.42}$$

Now we can find the optimal parameter $\mathbf{W}$ iteratively by the Newton-Raphson method based on both $\mathbf{H}_J$ and $\mathbf{g}_J$:

$$\begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_K \end{bmatrix}_{n+1} = \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_K \end{bmatrix}_n - \mathbf{H}_n^{-1}\, \mathbf{g}_n = \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_K \end{bmatrix}_n - \begin{bmatrix} \mathbf{H}_{11} & \cdots & \mathbf{H}_{1K} \\ \vdots & \ddots & \vdots \\ \mathbf{H}_{K1} & \cdots & \mathbf{H}_{KK} \end{bmatrix}_n^{-1} \begin{bmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_K \end{bmatrix}_n \tag{7.43}$$

Once $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_K]$ is available, any unlabeled sample $\mathbf{x}$ of unknown class can be classified into one of the K classes with the maximum conditional probability given in Eq. (7.29):

$$\text{if} \quad p(\hat{y} = k|\mathbf{x}) = \max_l \; p(\hat{y} = l|\mathbf{x}), \quad \text{then} \quad \mathbf{x} \in C_k \tag{7.44}$$

Whether we should use softmax regression or K logistic regressions for a problem of K classes $C_1, \cdots, C_K$ depends on the nature of the classes. The method of softmax regression is suitable if the K classes are mutually exclusive and independent, as assumed by the method. Otherwise, K logistic regression binary classifiers are more suitable.

Below is a Matlab function for estimating the K weight vectors in $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_K]$ based on the training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}$ and the hyperparameter λ:

```
function W=softmaxRegression(X,y,lambda)
    [d N]=size(X);
    K=length(unique(y));    % number of classes
    X=[ones(1,N); X];       % augmented data points
    d=d+1;
    Y=zeros(K,N);
    for n=1:N
        Y(y(n),n)=1;        % generate binary labeling Y
    end
    W=zeros(d*K,1);         % initial guess of K weight vectors
    I=eye(K);
    s=zeros(K,N);
    phi=zeros(K,N);         % softmax functions
    gi=zeros(d,1);          % ith gradient
    Hij=zeros(d,d);         % ij-th Hebbian
    er=9;
    tol=10^(-6);
    it=0;
    while er>tol
        it=it+1;
        W2=reshape(W,d,K);  % weight vectors in d x K 2-D array
        g=[];               % total gradient vector
        for i=1:K
            for n=1:N
                xn=X(:,n);  % get the nth sample
                t=0;
                for k=1:K
                    wk=W2(:,k); % get the kth weight vector
                    s(k,n)=exp(wk'*xn);
                    t=t+s(k,n);
                end
                phi(i,n)=s(i,n)/t; % softmax function
            end
            gi=X*(phi(i,:)-Y(i,:))'+lambda*W2(:,i);   % ith gradient
            g=[g; gi];      % stack all gradients into a long vector
        end
        H=[];               % total Hessian
        z=zeros(N,1);
        for i=1:K           % for the ith block row
            Hi=[];
            for j=1:K       % for the jth block in ith row
```

```
                for n=1:N
                   z(n)=phi(i,n)*((i==j)-phi(j,n));
                end
                Hij=X*diag(z)*X';
                Hi=[Hi Hij]; % append jth block
            end
            H=[H; Hi];       % append ith blow row
        end
        H=H+lambda*eye(d*K); % include regulation term
        W=W-inv(H)*g;        % update W by Newton's method
        er=norm(g);
    end
    W=reshape(W,d,K);        % reshape weight into d x K array
end
```

Here is the function for the classification of the unlabeled data samples in matrix $\mathbf{X}$ based on the weight vectors in $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_K]$:

```
function yhat=softmaxClassify(W,X)
    [d N]=size(X);       % dataset to be classified
    [d K]=size(W);       % model parameters
    X=[ones(1,N); X];    % augmented data points
    yhat=zeros(N,1);
    for n=1:N            % for each of the N samples
        xn=X(:,n);       % nth sample point
        t=0;
        for k=1:K
            wk=W(:,k);   % kth weight vector
            s(k)=exp(wk'*xn);
            t=t+s(k);
        end
        pmax=0;
        for k=1:K
            p=s(k)/t;    % probability based on softmax function
            if p>pmax
                kmax=k; pmax=p;
            end
        end
        yhat(n)=kmax;    % predicted class labeling
    end
end
```

Problems

1. Implement the method of logistic regression for binary classification, based on both (a) the gradient descent and (b) the Newton-Raphson methods. Then apply the code to the datasets below:
   - dataset 1 /e176/homework/2ClassData.txt
   - dataset 2 /e176/homework/2CocentricData.txt
   - dataset 3 /e176/homework/XOR.txt

   Each data file contains three columns for $x_1,\ x_2,\ y$, respectively. Here y is the binary labeling (1 or -1), indicating to which of the two classes point $\mathbf{x} = [x_1,\ x_2]^T$ belongs.

   Compare the performances of the two methods in terms of the number of iterations and percentage error.

   Visualize your classification results by plotting all 2-D data points $\mathbf{x}_n$ in the training set and coloring them first based on their labeling y_n (0 or 1) to show their true class identity (e.g., red for class 1 and blue for class 2), and then based on your classification result y'_n. Comparing these two plots you will see how effective your classification method is.

   Show your classification results in the form of a 2×2 confusion matrix, where the entry on the ith row and jth column $(i, j = 1, 2)$ is the number data points labeled by y to belong to the ith class but classified to the jth class. Ideally, if all data points are classified correctly, the confusion matrix is diagonal. Also give the percentage error, defined as the ratio of the number of mis-classified data points and the total number of data points.
2. Develop Matlab code for multi-class classification based on both (a) one-vs-one method and (b) one-vs-rest method, based on logistic regression for binary classification, then carry out classification to the following 3-class datasets. Compare their results of the two methods in terms of execution time and error rates.
   - dataset 1 in /e176homework/3ClassData1.txt

     The first three columns are for the components for each data point $\mathbf{x}_n = [x_{1n},\ x_{2n},\ x_{3n}]^T$, and the fourth column is the corresponding labeling y_n, indicating to which of the $K = 3$ classes $\mathbf{x}_n$ belongs.
   - dataset 2 in /e176/homework/3ClassData2.txt

     The first two columns are for the components for each data point $\mathbf{x}_n = [x_{1n},\ x_{2n}]^T$, and the third column is the corresponding labeling y_n, indicating to which of the $K = 3$ classes $\mathbf{x}_n$ belongs.
   - The iris dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set).

     This dataset can be read into your code by Matlab command `x=iris_dataset`, where `x` is a 4×150 matrix, each of the 150 columns is for one of the 150 data samples, containing 50 samples for each of the three species of Iris (Iris setosa, Iris virginica and Iris versicolor).

   Show your classification results in the form of a confusion matrix, of which the entry in the ith row and jth column is the number of data points in the

training set belonging to the ith class according to the labeling y but classified into the jth class by your classifier. Also give the percentage error, defined as the number of data samples mis-classified over the total number of samples. Compare the percentage error of three methods above for each of the three datasets.

Visualize both the training dataset (the ground truth) and your classification results in 3-D space (using Matlab function `scatter3`), color code each data point $\mathbf{x}_n$ red, green or blue according to its ground truth class labeling y_n. Repeat the process but now with each data point $\mathbf{x}_n$ color coded according to the classification result y'_n.

3. Develop Matlab code to Implement the softmax regression method for multiclass classification, and apply it to the datasets in the previous problem. Show your results in confusion matrix and visualization. Compare your results with those obtained in the previous problem.

8 Gaussian Process Regression and Classification

8.1 Gaussian Process Regression

The *Gaussian process regression (GPR)* is yet another regression method that fits a regression function $f(\mathbf{x})$ to the data samples in the given training set. Different from all previously considered algorithms that treat the regression model as a deterministic function $f(\mathbf{x})$ that takes an explicitly specified form, the GPR treats the regression function $f(\mathbf{x})$ as a stochastic *Gaussian process (GP)*, of which the value $f(\mathbf{x})$ at any point $\mathbf{x}$ is assumed to be a random variable with a Gaussian distribution.

We can therefore assume the joint probability density distribution (pdf) of $\mathbf{f}(\mathbf{X}) = [f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N)]^T$ for all i.i.d. samples in the training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ takes the following form:

$$p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{m}_f,\ \mathbf{\Sigma}_f) = \mathcal{N}(\mathbf{0},\ \mathbf{\Sigma}_f) \tag{8.1}$$

where the mean vector $\mathbf{m}_f = \mathbf{0}$ is assumed to be zero for convenience and without loss of generality. Based on this prior probability of the regression function $f(\mathbf{x})$, we desire to find the posterior $p(\mathbf{f}_*|\mathcal{D}, \mathbf{X}_*)$ for the function of any unlabeled test data $\mathbf{X}_*$ given the training set $\mathcal{D}$.

We also assume the observed data $\mathbf{y} = \mathbf{f}(\mathbf{x}) + \mathbf{n}$ is contaminated by noise $\mathbf{n}$, with a zero-mean Gaussian pdf

$$p(\mathbf{n}) = \mathcal{N}(\mathbf{0},\ \sigma_n^2\mathbf{I}) \tag{8.2}$$

where the diagonal covariance matrix $\sigma_n^2\mathbf{I}$ is based on the assumption that all components of $\mathbf{n}$ are independent of each other. We further assume noise $\mathbf{n}$ is independent of $\mathbf{f}$, i.e., $\text{Cov}[\mathbf{y}, \mathbf{n}] = \mathbf{0}$. Now $\mathbf{y} = \mathbf{f}(\mathbf{X}) + \mathbf{n}$, as the sum of two zero-mean Gaussian variables, is also a zero mean Gaussian process.

The unknown covariance matrix $\mathbf{\Sigma}_f$ in Eq. (8.1) can be constructed based on the assumption that the regression function $f(\mathbf{x})$ is smooth, in the sense that any two function values $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ are more correlated if the distance $||\mathbf{x}_m - \mathbf{x}_n||$ between the two corresponding data points is small, but less so if the distance is large. Now the covariance of any two function values $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ can be modeled by the squared exponential (SE) function, also called the

kernel function:

$$\text{Cov}[f(\mathbf{x}_m), f(\mathbf{x}_n)] = \exp\left(-\frac{1}{\alpha^2}||\mathbf{x}_m - \mathbf{x}_n||^2\right) = k(\mathbf{x}_m, \mathbf{x}_n) = k(\mathbf{x}_n, \mathbf{x}_m) \quad (8.3)$$

This SE function may take any value between 1 when $||\mathbf{x}_m - \mathbf{x}_n|| = 0$ and $f(\mathbf{x}_m) = f(\mathbf{x}_n)$ are maximally correlated, and 0 when $||\mathbf{x}_m - \mathbf{x}_n|| = \infty$ and $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ are least correlated.

Here α is a hyperparameter that controls the smoothness of $f(\mathbf{x})$. To see this, we consider two extreme cases. First, if $\alpha \to \infty$, then the SE function approaches $e^0 = 1$ and $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ are maximally correlated, i.e., $f(\mathbf{x})$ is alsolutely smooth. Second, if $\alpha \to 0$, then the SE function approaches $e^{-\infty} = 0$ and $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ are completely uncorrelated, i.e., $f(\mathbf{x})$ is least smooth. We can set α to any value between 0 and 1 depending on the desired smoothness.

The smoothness of the function $f(\mathbf{x})$ plays an important role as it determines whether $f(\mathbf{x})$ over or underfits the sample points in the training data. If $f(\mathbf{x})$ is not smooth enough, it may be affected by outlier samples due to noise and overfitting the data; on the other hand, if $f(\mathbf{x})$ is too smooth, it may not reflect the valid signal variations and underfits the data. By adjusting α, a proper tradeoff between over and underfittings can be made.

Given $\text{Cov}[f(\mathbf{x}_m), f(\mathbf{x}_n)] = k(\mathbf{x}_m, \mathbf{x}_n)$ in Eq. (8.3), the covariance matrix $\boldsymbol{\Sigma}_f$ in Eq. (8.1) can be expressed as:

$$\boldsymbol{\Sigma}_f = \text{Cov}[\mathbf{f}, \mathbf{f}] = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & \cdots & k(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix} = K(\mathbf{X}, \mathbf{X}) = \mathbf{K} \quad (8.4)$$

where $\mathbf{K}$ is the kernel matrix used to model $\boldsymbol{\Sigma}_f$. Note that the value of all diagonal components are the same $k(\mathbf{x}_n, \mathbf{x}_n) = 1$, and the farther away an off-diagonal component $k(\mathbf{x}_m, \mathbf{x}_n)$ is from the diagonal (with greater $||\mathbf{x}_m - \mathbf{x}_n||$), the lower value it takes. In general, the off-diagonal components of $\boldsymbol{\Sigma}_f$ have larger values for a smooth function $f(\mathbf{x})$ than those for a less smooth function.

Now that the prior $p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_f)$ is constructed based on training set $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, we can further find the posterior of $\mathbf{f}_* = \mathbf{f}(\mathbf{X}_*)$ for any test dataset $\mathbf{X}_* = [\mathbf{x}_{1*}, \cdots, \mathbf{x}_{M*}]$ containing M unlabeled samples. In the following, we will consider first the special noise-free case with $\mathbf{n} = \mathbf{0}$ and $\mathbf{y} = \mathbf{f} + \mathbf{n} = \mathbf{f}$, and then the general case of non-zero noise case with $\mathbf{f} \neq \mathbf{y}$.

In the noise-free case, as both $\mathbf{f} = \mathbf{y}$ and $\mathbf{f}_*$ are samples from the same Gaussian process, and their joint pdf conditioned on $\mathbf{X}_*$ as well as $\mathbf{X}$ can be expressed as

$$\begin{aligned} p(\mathbf{f}, \mathbf{f}_*|\mathbf{X}, \mathbf{X}_*) &= p\left(\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \middle| \mathbf{X}, \mathbf{X}_*\right) = \mathcal{N}\left(\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_f & \boldsymbol{\Sigma}_{ff_*} \\ \boldsymbol{\Sigma}_{f_*f} & \boldsymbol{\Sigma}_{f_*} \end{bmatrix}\right) \\ &= \mathcal{N}\left(\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{bmatrix}\right) \end{aligned} \quad (8.5)$$

where $\boldsymbol{\Sigma}_f$ is given in Eq. (8.4), and the other three covariance matrices can be

simlarly modeled by their corresponding kernel matrices:

$$\begin{cases} \boldsymbol{\Sigma}_{ff_*} = \text{Cov}[\mathbf{f}, \mathbf{f}_*] = \mathbf{K}_* = K(\mathbf{X}, \mathbf{X}_*) \\ \boldsymbol{\Sigma}_{f_*f} = \text{Cov}[\mathbf{f}_*, \mathbf{f}] = \mathbf{K}_*^T = K(\mathbf{X}_*, \mathbf{X}) \\ \boldsymbol{\Sigma}_{f_*} = \text{Cov}[\mathbf{f}_*, \mathbf{f}_*] = \mathbf{K}_{**} = K(\mathbf{X}_*, \mathbf{X}_*) \end{cases} \tag{8.6}$$

Given this joint pdf $p(\mathbf{f}, \mathbf{f}_*|\mathbf{X}, \mathbf{X}_*)$ of both $\mathbf{f}$ and $\mathbf{f}_*$ in Eq. (8.5), we can further find the conditional pdf $p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*)$ of $\mathbf{f}_*$ given $\mathbf{f}$ as well as $\mathbf{X}$ and $\mathbf{X}_*$ (see properties of the Gaussian distributions in Section B.1.5):

$$p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*) = \mathcal{N}(\mathbf{m}_{f_*|f}, \boldsymbol{\Sigma}_{f_*|f}) \tag{8.7}$$

with

$$\begin{cases} \mathbf{m}_{f_*|f} &= \text{E}[\mathbf{f}_*|\mathbf{f}] = \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{f} \\ \boldsymbol{\Sigma}_{f_*|f} &= \text{Cov}[\mathbf{f}_*|\mathbf{f}] = \mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{K}_* \end{cases} \tag{8.8}$$

Based on the posterior in Eq. (8.7), we can estimate $f(\mathbf{x}_*)$ at any unlabeled sample in the test set $\mathbf{X}_*$ in the noise-free case.

We next consider the noisy case with $\mathbf{n} \neq \mathbf{0}$ and $\mathbf{y} = \mathbf{f} + \mathbf{n} \neq \mathbf{f}$. Now we need to find a new posterior $p(\mathbf{f}_*|\mathbf{y}, \mathbf{X}, \mathbf{X}_*) = p(\mathbf{f}_*|\mathbf{X}_*, \mathcal{D})$ in terms of $\mathbf{y}$ in the training set $\mathcal{D}$ to replace $p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*)$ in Eq. (8.7) in terms of $\mathbf{f}$. To do so, we follow the same process above to find first the joint pdf of $\mathbf{f}_*$ and $\mathbf{y}$ and then the conditional pdf of $\mathbf{f}_*$ given $\mathbf{y}$.

The joint pdf of $\mathbf{f}_*$ and $\mathbf{y}$ (both zero-mean Gaussian) can be expressed as:

$$p(\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix}|\mathbf{X}, \mathbf{X}_*) = \mathcal{N}\left(\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_y & \boldsymbol{\Sigma}_{yf_*} \\ \boldsymbol{\Sigma}_{f_*y} & \boldsymbol{\Sigma}_{f_*} \end{bmatrix}\right) \tag{8.9}$$

where $\boldsymbol{\Sigma}_{f_*}$ is given in Eq. (8.6) and

$$\begin{aligned} \boldsymbol{\Sigma}_y &= \text{Cov}[\mathbf{y}, \mathbf{y}] = \text{Cov}[\mathbf{f} + \mathbf{n}, \mathbf{f} + \mathbf{n}] = \text{Cov}[\mathbf{f}, \mathbf{f}] + 2\ \text{Cov}[\mathbf{f}, \mathbf{n}] + \text{Cov}[\mathbf{n}, \mathbf{n}] \\ &= \text{Cov}[\mathbf{f}, \mathbf{f}] + \text{Cov}[\mathbf{n}, \mathbf{n}] = \boldsymbol{\Sigma}_f + \boldsymbol{\Sigma}_n = \mathbf{K} + \sigma_n^2\mathbf{I} \\ \boldsymbol{\Sigma}_{yf_*} &= \text{Cov}[\mathbf{y}, \mathbf{f}_*] = \text{Cov}[\mathbf{f} + \mathbf{n}, \mathbf{f}_*] = \text{Cov}[\mathbf{f}, \mathbf{f}_*] = \boldsymbol{\Sigma}_{ff_*} = \mathbf{K}_* \\ \boldsymbol{\Sigma}_{f_*y} &= \boldsymbol{\Sigma}_{yf_*}^T = \mathbf{K}_*^T \end{aligned} \tag{8.10}$$

where we have used the fact that $\text{Cov}[\mathbf{f}, \mathbf{n}] = \text{Cov}[\mathbf{f}_*, \mathbf{n}] = \mathbf{0}$ as noise $\mathbf{n}$ is assumed to be independent of $\mathbf{f}$. Now we have

$$\text{Cov}\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Sigma}_y & \boldsymbol{\Sigma}_{yf_*} \\ \boldsymbol{\Sigma}_{f_*y} & \boldsymbol{\Sigma}_{f_*} \end{bmatrix} = \begin{bmatrix} \mathbf{K} + \sigma_n^2\mathbf{I} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{bmatrix} \tag{8.11}$$

We then get the desired posterior, the conditional pdf of $\mathbf{f}_*$ given $\mathbf{X}_*$ as well as the training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$:

$$p(\mathbf{f}_*|\mathbf{X}_*, \mathcal{D}) = p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(\mathbf{m}_{f_*}, \boldsymbol{\Sigma}_{f_*}) \tag{8.12}$$

with the mean and covariance

$$\begin{cases} \mathbf{m}_{f_*} &= \text{E}[\mathbf{f}_*|\mathbf{y}] = \mathbf{K}_*^T\left(\mathbf{K} + \sigma_n^2\mathbf{I}\right)^{-1}\mathbf{y} \\ \boldsymbol{\Sigma}_{f_*} &= \text{Cov}[\mathbf{f}_*|\mathbf{y}] = \mathbf{K}_{**} - \mathbf{K}_*^T\left(\mathbf{K} + \sigma_n^2\mathbf{I}\right)^{-1}\mathbf{K}_* \end{cases} \tag{8.13}$$

We see that these expressions are the same as those in Eq. (5.74) previously obtained by Bayesian linear regression, i.e., the same results can be derived by two very different methods.

In the special noise-free case when $\mathbf{n} = \mathbf{0}$ and $\sigma_n = 0$, the mean and covariance in Eq. (8.13) become the same as those in Eq. (8.8), and as $\mathbf{y} = \mathbf{f} + \mathbf{n} = \mathbf{f}$, the posterior in Eq. (8.12) becomes the same as that in Eq. (8.7):

$$p(\mathbf{f}_*|\mathbf{X}_*, \mathcal{D}) = p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) = p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{f}) \tag{8.14}$$

The Matlab code below is the essential part of the Gaussian process regression algorithm. It takes as the input the training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$, the test samples in $\mathbf{X}_*$, and the noise variance σ_n^2, and it generates as the output the posterior $p(\mathbf{f}_*|\mathcal{D}, \mathbf{X}_*)$, in terms of its mean as the GPR regression function, and its covariance matrix, of which the variances on the diagonal represent the confidence or certainty of the regression.

```
xstar=xmin:0.1:xmax;           % domain of function
n=length(xstar);               % number of test samples
Kss=Kernel(xstar,xstar);       % K(x*,x*), covariance of prior
[L p]=chol(Kss);               % Cholesky decomposition
for i=1:k
    plot(xstar,L'*randn(n,1))  % plot sample curves from prior
    hold on
end
hold off
x=xstar(randi(n,1,N));           % get N random samples from xstar
y=f(xstar)+randn(1,n)*sigma_n^2;  % get their noisy labeling values
Kxx=Kernel(x,x)+sigma_n^2*eye(N); % K(x,x)
Kxs=Kernel(x,xstar);              % K(x,x*)
Ksx=Kxs';                      % K(x*,x)
fmean=Ksx*inv(Kxx)*y;          % posterior mean, regression function
fcov=Kss-Ksx*inv(Kxx)*Kxs;     % covariance of posterior
plot(xstar,fmean,xstar,fmean+diag(fcov),xstar,fmean-diag(fcov));
[L p]=chol(fcov);              % Cholesky decomposition
for i=1:k
    plot(xstar,L'*randn(n,1))  % plot sample curves drawn from posterior
    hold on
end
hold off
```

Here `Kernel` is a function that generates the squared exponential kernel matrix based on two sets of data samples in `X1` and `X2`:

```
function K=Kernel(X1,X2)
    a=1;            % parameter that controls smoothness of f(x)
```

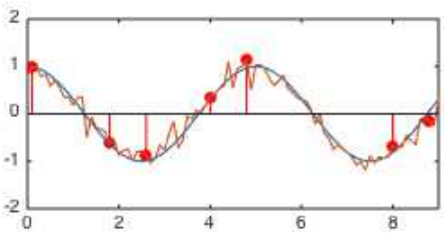


Figure 8.1 A Noisy Sinusoid

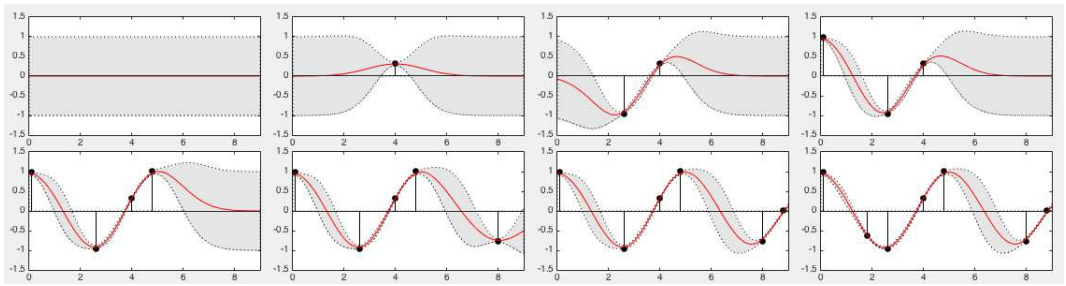


Figure 8.2 Gaussian Process Regression with Varying Data Points

```
    m=length(X1);
    n=length(X2);
    K=zeros(m,n);
    for i=1:m
        for j=1:n
            d=norm(X1(:,i)-X2(:,j));
            K(i,j)=exp(-(d/a)^2);
        end
    end
end
```

Example 8.1 As shown in Fig. 8.1, a sinusoidal function $f(x) = \cos(2\pi f x)$ (with $f = 0.2$) contaminated by Gaussian noise with $p(n) = \mathcal{N}(0, \sigma_r^2)$ ($\sigma_r = 0.2$) is sampled at $N = 7$ random positions over the duration between the limits $x_{min} = 0$ and $x_{max} = 9$.. These samples form the training dataset $\mathcal{D} = \{(x_n, y_n)\,(n = 1, \cdots, N)\}$.

The method of GPR is used to estimate the mean and variance of $f_{i*} = f(x_{i*})$ at 91 test points x_{i*}, $(i = 1, \cdots, 91)$, evenly distributed over the entire duration. The 8 panels of Fig. 8.2 show the estimated mean $\mathrm{E}[f_*]$, and the variance $\pm\,\mathrm{Var}[f_*]$, both above and below the mean, to form a region of uncertainty of the estimates. The estimated function values are more reliable in the region that is narrow but less so if it is wide.

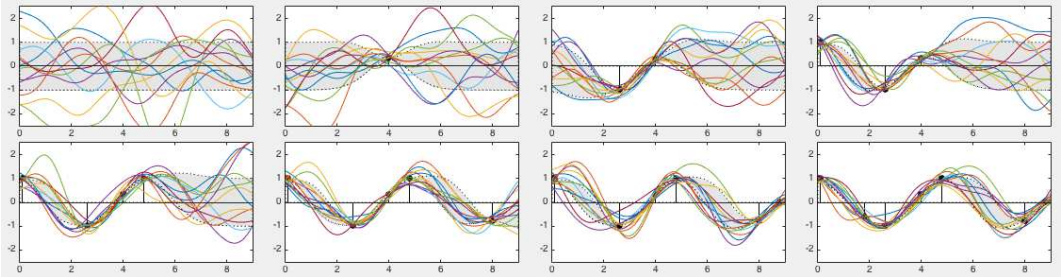


Figure 8.3 Samples Drawn from Posteriors

The top left panel represents the prior probability $p(f_*|x_*)$ of the estimated model output $f_* = \hat{y}_*$ at point x_* with no knowledge based on the training set $\mathbf{x}$ ($N = 0$ training samples), which is assumed to have zero-mean $\mathrm{E}[f_*] = 0$ and unity variance $\mathrm{Var}[f(x_*)] = k(x_*, x_*) = 1$. The subsequent seven panels show the mean and variance of the posterior $p(f_* = \hat{y}_*|x_*, \mathcal{D})$ estimated based on $N = 1, \cdots, 7$ training samples in $\mathcal{D}$. Note that wherever an additional training sample x is available, the estimated mean $\mu_{f_*|y}$ is modified to be close to the true function value $f(x)$, and the region of uncertainty is narrowed due to the reduced variance of the posterior, i.e., the uncertainty of the estimate is reduced, or the confidence is increased. As shown in the last panel, when all $N = 7$ training samples are used, the general shape of the original function is reasonably reconstructed.

The 8 panels in Fig. 8.3 show 12 sample curves randomly drawn from each posterior $p(f_*|\mathbf{x}_*, \mathcal{D})$. They can be treated as different interpolations for the N samples in $\mathcal{D}$, and any of them can be used to predict the outputs y_* at any input points x_*. The mean of these samples shown as the red curve is the same as $\mathrm{E}[f_*]$ shown in the previous figure.

We see that wherever a training sample $f(x)$ is available at x, the variance and thereby the width of the region of uncertainty is reduced, and correspondingly the sample curves in the neighborhood of x are more tightly distributed. Specifically, we see that the sample curves drawn from the prior in the top left panel are very loosely distributed, while those samples drawn from the posterior obtained based on all $N = 7$ training points in the bottom right panel are relatively tightly distributed. In particular, note that due to lack of training data in the interval $5 < x < 8$, the region of uncertainty is significantly wider than else where, and the sample curves are much more loosely located.

The parameter α of the kernel function plays an important role in the GPR results, as shown in the three panels in Fig. 8.4, corresponding to $\alpha = 0.5, 1.5, 3.0$. When α is too small (top panel), the model may overfit the data, as the regression function fits the training samples well but it not smooth and it does not interpolate regions between the training samples well (with large variance); one the other hand, when α is too large (bottom panel), the regression function is

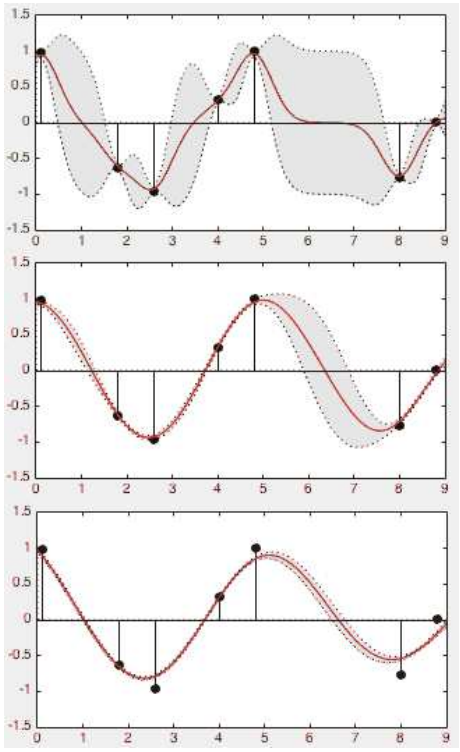


Figure 8.4 Over Fitting (top) and Under Fitting (bottom) by GPR

smooth but it may underfit the data, as it does not fit the training samples well. By adjusting the value of α carefully we can makea a proper tradeoff between overfitting and underfitting as shown in the middle panel.

8.2 Gaussian Process Classifier – Binary

In both logistic regression and softmax regression considered previously, the linear regression function $\hat{y} = f(\mathbf{x}) = \mathbf{x}^T\mathbf{w}$ is converted to the probability $p(\hat{y} = k|\mathbf{x}, \mathbf{W})$ for $\mathbf{x} \in C_k$, by either the logistic function $\sigma(f(\mathbf{x}))$ for binary classification, or the softmax function $s(f(\mathbf{x}))$ for multiclass classification. Also, in Gaussian process regression (GPR), the regression function $f(\mathbf{x})$ is treated as a Gaussian process. Now we consider the *Gaussian process classification (GPC)* based on the combination of both logistic/softmax regression and GPR. We will consider binary GPC based on the logistic function in this section, and then multiclass GPC based on softmax function in the following section.

Here in binary GPC, we assume the training samples are labeled by 1 or -1 (instead of 0) for mathematical convenience, i.e., $\mathbf{x}_n \in C_+$ if $y_n = 1$ or $\mathbf{x}_n \in C_-$ if $y_n = -1$. Similar to logistic regression, here we also convert the regression

function $f(\mathbf{x})$, treated as a Gaussian process, into the probability for $\mathbf{x} \in C_k$ by the logistic function $\sigma(f(\mathbf{x}))$, which, as a function of a random argument $f(\mathbf{x})$, is also random. We therefore define $p(y = 1|\mathbf{x}, \mathcal{D})$ as the expectation of $\sigma(f(\mathbf{x}))$ with respect to $f(\mathbf{x})$:

$$p(y = 1|\mathbf{x}, \mathcal{D}) = E_f[\sigma(f(\mathbf{x}))] = \int \sigma(f(\mathbf{x}))\, p(f|\mathcal{D})\, df \tag{8.15}$$

As function $f(\mathbf{x})$ is marginalized (averaged out) by this integral, it is hidden instead of explicitly specified, and is therefore called a *latent function*. This $p(y = 1|\mathbf{x}, \mathcal{D})$ can be generalized for multiple samples in the training set $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$:

$$p(\mathbf{y} = 1|\mathcal{D}) = E_f[\sigma(\mathbf{f}(\mathbf{X}))] = \int \sigma(\mathbf{f}(\mathbf{X}))\, p(\mathbf{f}|\mathcal{D})\, d\mathbf{f} \tag{8.16}$$

Here the posterior $p(\mathbf{f}|\mathcal{D})$ of $\mathbf{f}$ given $\mathcal{D}$ can be found based on Bayes' theorem in terms of the likelihood $p(\mathbf{y}|\mathbf{f}) = L(\mathbf{f}|\mathbf{y})$ and the prior $p(\mathbf{f}|\mathbf{X})$:

$$p(\mathbf{f}|\mathcal{D}) = p(\mathbf{f}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{y}, \mathbf{f}|\mathbf{X})}{p(\mathbf{y}|\mathbf{X})} = \frac{p(\mathbf{y}|\mathbf{f})\, p(\mathbf{f}|\mathbf{X})}{p(\mathbf{y}|\mathbf{X})} \propto p(\mathbf{y}|\mathbf{f})\, p(\mathbf{f}|\mathbf{X}) \tag{8.17}$$

As always, the denominator $p(\mathbf{y}|\mathbf{X})$ independent of $\mathbf{f}$ is dropped.

We first find the likelihood $p(\mathbf{y}|\mathbf{f})$. The Gaussian process $f(\mathbf{x})$ is mapped by the logistic function into the probability for $y = 1$ representing $\mathbf{x} \in C_+$, or $y = -1$ representing $\mathbf{x} \in C_-$:

$$\begin{aligned} p(y = 1|f(\mathbf{x})) &= \sigma(f(\mathbf{x})) \\ p(y = -1|f(\mathbf{x})) &= 1 - p(y = 1|f(\mathbf{x})) = 1 - \sigma(f(\mathbf{x})) = \sigma(-f(\mathbf{x})) \end{aligned} \tag{8.18}$$

These two cases can be combined to get the conditional probability of y given $f(\mathbf{x})$:

$$p(y|f) = \sigma(y\, f) = \frac{1}{1 + e^{-yf}} = \frac{e^{yf}}{1 + e^{yf}} \tag{8.19}$$

The likelihood of $\mathbf{f}$ for all N i.i.d. samples in the training set $\mathbf{X}$ is:

$$p(\mathbf{y}|\mathbf{f}) = \prod_{n=1}^{N} p(y_n|f_n) = \prod_{n=1}^{N} \sigma(y_n\, f_n) \tag{8.20}$$

We then consider the prior $p(\mathbf{f}|\mathbf{X})$, which is assumed to be a zero-mean Gaussian $p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_f)$, where the covariance matrix $\mathbf{\Sigma}_f$ is constructed based on the training set $\mathbf{X}$, the same as in GPR. Specifically, the covariance between $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ in the mth row and nth column of $\mathbf{\Sigma}_f$ is modeled by the squared exponential (SE) kernel:

$$\mathrm{Cov}[f(\mathbf{x}_m), f(\mathbf{x}_n)] = k(\mathbf{x}_m, \mathbf{x}_n) = \exp\left(-\frac{1}{a^2}||\mathbf{x}_m - \mathbf{x}_n||^2\right), \qquad (m, n = 1, \cdots, N) \tag{8.21}$$

Such a covariance matrix is desired for GPC as well as GPR, so that $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ are more correlated if $\mathbf{x}_m$ and $\mathbf{x}_n$ are close together, but less so if they are

farther apart. However the justification for such a property is different. In GPR, this property is desired for the smoothness of the regression function; while here in GPC, this property is also desired so that function values $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ for two samples $\mathbf{x}_m$ and $\mathbf{x}_n$ close to each other in the feature space are more correlated and therefore the two samples are more likely to be classified into the same class, but less so if they are far apart.

Also, as discussed in GPR, here the parameter a in the SE controls the smoothness of the function. If $a \to \infty$, the value of the SE approaches 1, then $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ are highly correlated and $f(\mathbf{x})$ is very smooth; but if $a \to 0$, SE approaches 0, then $f(\mathbf{x}_m)$ and $f(\mathbf{x}_n)$ are not correlated and $f(\mathbf{x})$ is no longer smooth. By adjusting a, a proper tradeoff can be made between overfitting and underfitting.

Having found both the likelihood $p(\mathbf{y}|\mathbf{f})$ and the prior $p(\mathbf{f})$, we can get the posterior in Eq. (8.17):

$$\begin{aligned} p(\mathbf{f}|\mathcal{D}) &\propto p(\mathbf{y}|\mathbf{f})\, p(\mathbf{f}|\mathbf{X}) = p(\mathbf{y}|\mathbf{f})\, \mathcal{N}(\mathbf{0}, \mathbf{K}) \\ &= \prod_{n=1}^{N} \sigma(y_n\, f_n)\ \frac{1}{(2\pi)^{d/2}|\mathbf{K}|^{1/2}} \exp\left(-\frac{1}{2}\mathbf{f}^T\mathbf{K}^{-1}\mathbf{f}\right) \\ &\propto \prod_{n=1}^{N} \sigma(y_n\, f_n)\ \exp\left(-\frac{1}{2}\mathbf{f}^T\mathbf{K}^{-1}\mathbf{f}\right) \end{aligned} \tag{8.22}$$

Note that the likelihood $p(\mathbf{y}|\mathbf{f})$ is not Gaussian, as the binary labeling $\mathbf{y}$ of the training set $\mathbf{X}$ is not continuous. Consequently, the posterior $p(\mathbf{f}|\mathcal{D})$, as a product of the Gaussian prior and non-Gaussian likelihood, is not Gaussian. However, we can carry out *Laplace approximation* and still approximate the posterior as a Gaussian:

$$p(\mathbf{f}|\mathcal{D}) \approx \mathcal{N}(\mathbf{m}_{f|D}, \mathbf{\Sigma}_{f|D}) \tag{8.23}$$

of which the mean $\mathbf{m}_{f|D}$ and covariance $\mathbf{\Sigma}_{f|D}$ are to be obtained in the following. We further get the log posterior denoted by $\psi(\mathbf{f})$:

$$\psi(\mathbf{f}) = \log\, p(\mathbf{f}|\mathcal{D}) = \sum_{n=1}^{N} \log \sigma(y_n\, f_n) - \frac{N}{2}\log(2\pi) - \frac{1}{2}\log|\mathbf{K}| - \frac{1}{2}\mathbf{f}^T\mathbf{K}^{-1}\mathbf{f} \tag{8.24}$$

The two middle terms independent of $\mathbf{f}$ are dropped.

Now that the posterior $p(\mathbf{f}|\mathcal{D})$ is approximated as a Gaussian, we can find the gradient vector $\mathbf{g}_\psi(\mathbf{f})$ and Hessian matrix $\mathbf{H}_\psi(\mathbf{f})$ of the log posterior:

$$\mathbf{g}_\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\psi(\mathbf{f}) = -\mathbf{\Sigma}_{f|D}^{-1}(\mathbf{f} - \mathbf{m}_{f|D}) \tag{8.25}$$

$$\mathbf{H}_\psi(\mathbf{f}) = \frac{d^2}{d\mathbf{f}^2}\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\mathbf{g}_\psi(\mathbf{f}) = -\mathbf{\Sigma}_{f|D}^{-1} \tag{8.26}$$

On the other hand, we can also find $\mathbf{g}_\psi(\mathbf{f})$ and $\mathbf{H}_\psi(\mathbf{f})$ by directly taking the first and second order derivatives of the log posterior::

$$\mathbf{g}_\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\ \log\, p(\mathbf{y}|\mathbf{f})\ - \frac{d}{d\mathbf{f}}\left(\frac{1}{2}\mathbf{f}^T\mathbf{K}^{-1}\mathbf{f}\right) = \mathbf{w} - \mathbf{K}^{-1}\mathbf{f} \tag{8.27}$$

$$\mathbf{H}_\psi(\mathbf{f}) = \frac{d^2}{d\mathbf{f}^2}\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\mathbf{g}_\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\left(\mathbf{w} - \mathbf{K}^{-1}\mathbf{f}\right) = \mathbf{W} - \mathbf{K}^{-1} \tag{8.28}$$

where we have defined

$$\mathbf{w} = \frac{d}{d\mathbf{f}}\log p(\mathbf{y}|\mathbf{f}), \qquad \mathbf{W} = \frac{d^2}{d\mathbf{f}^2}\log p(\mathbf{y}|\mathbf{f}) = \frac{d\mathbf{w}}{d\mathbf{f}} \tag{8.29}$$

which are respectively the gradient vector and Hessian matrix of the log function

$$\log p(\mathbf{y}|\mathbf{f}) = \log \prod_{n=1}^{N} \sigma(y_n\, f_n) \;= \sum_{n=1}^{N} \log \sigma(y_n\, f_n) \;= \sum_{n=1}^{N} \log\left(\frac{1}{1+\exp(-y_n f_n)}\right) \tag{8.30}$$

We first find the first and second order derivatives with respect to a single component f_n of $\mathbf{f} = [f_1, \cdots, f_N]^T$:

$$\begin{aligned}
\frac{d}{df_n}\log\, p(y_n|f_n) &= \frac{d}{df_n}\,\log\sigma(y_n f_n) = \frac{d}{df_n}\,\log\left(\frac{1}{1+\exp(-y_n f_n)}\right) = \frac{y_n}{1+\exp(y_n f_n)} \\
\frac{d^2}{df_n^2}\log\, p(y_n|f_n) &= \frac{d^2}{df_n^n}\log\,\sigma(y_n f_n) = \frac{d}{df_n}\left(\frac{y_n}{1+\exp(y_n f_n)}\right) = \frac{-\exp(-y_n f_n)}{(1+\exp(-y_n f_n))^2}
\end{aligned} \tag{8.31}$$

where we have used the facts that $y_n = \pm 1$ and $y_n^2 = 1$. We then find

$$\begin{aligned}
\mathbf{w} = \frac{d}{d\mathbf{f}}\log p(\mathbf{y}|\mathbf{f}) &= \begin{bmatrix} d/df_1 \\ \vdots \\ d/df_N \end{bmatrix} \sum_{n=1}^{N} \log p(y_n|f_n) \\
&= \begin{bmatrix} d\log p(y_1|f_1)/df_1 \\ \vdots \\ d\log p(y_N|f_N)/df_N \end{bmatrix} = \begin{bmatrix} \frac{y_1}{1+e^{y_1 f_1}} \\ \vdots \\ \frac{y_N}{1+e^{y_N f_N}} \end{bmatrix}
\end{aligned} \tag{8.32}$$

and

$$\begin{aligned}
\mathbf{W} = \frac{d^2}{d\mathbf{f}^2}\log p(\mathbf{y}|\mathbf{f}) &= \begin{bmatrix} \frac{\partial^2}{\partial f_1 \partial f_1} & \cdots & \frac{\partial^2}{\partial f_1 \partial f_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial f_N \partial f_1} & \cdots & \frac{\partial^2}{\partial f_N \partial f_N} \end{bmatrix} \sum_{n=1}^{N} \log p(y_n|f_n) \\
&= \mathrm{diag}\left[\frac{d^2}{df_1^2}\log p(y_1|f_1), \cdots, \frac{d^2}{df_N^2}\log p(y_N|f_N)\right] \\
&= \mathrm{diag}\left[\frac{-e^{-y_1 f_1}}{(1+e^{-y_1 f_1})^2}, \cdots, \frac{-e^{-y_N f_N}}{(1+e^{-y_N f_N})^2}\right]
\end{aligned} \tag{8.33}$$

Equating the two expressions for $\mathbf{H}_\psi(\mathbf{f})$ in Eqs. (8.26) and (8.28) we get:

$$\mathbf{H}_\psi(\mathbf{f}) = \mathbf{W} - \mathbf{K}^{-1} = -\boldsymbol{\Sigma}_{f|D}^{-1}, \qquad \text{i.e.,} \qquad \boldsymbol{\Sigma}_{f|D} = (\mathbf{K}^{-1} - \mathbf{W})^{-1} \tag{8.34}$$

Equating the two expressions for $\mathbf{g}_\psi(\mathbf{f})$ in Eqs. (8.25) and (8.27) we get:

$$\begin{aligned}
\mathbf{g}_\psi(\mathbf{f}) &= -\boldsymbol{\Sigma}_{f|D}^{-1}(\mathbf{f} - \mathbf{m}_{f|D}) = \mathbf{w} - \mathbf{K}^{-1}\mathbf{f} \\
\text{i.e.,} \qquad \mathbf{m}_{f|D} &= \mathbf{f} + \boldsymbol{\Sigma}_{f|D}\left(\mathbf{w} - \mathbf{K}^{-1}\mathbf{f}\right) = \mathbf{f} - \mathbf{H}_\psi^{-1}\mathbf{g}_\psi
\end{aligned} \tag{8.35}$$

Substituting $\mathbf{\Sigma}_{f|D}$ given in Eq. (8.34) into the equation and solving it for $\mathbf{m}_{f|D}$, we get

$$\mathbf{m}_{f|D} = \mathbf{f} + (\mathbf{K}^{-1} + \mathbf{W})^{-1}(\mathbf{w} - \mathbf{K}^{-1}\mathbf{f}) \tag{8.36}$$

We recognize this is actually the iteration of Newton's method solving equation $\mathbf{g}_\psi(\mathbf{f}) = \mathbf{0}$, in consistence with the fact that when $\mathbf{f} = \mathbf{m}_{f|D}$, the log posterior $\psi(\mathbf{f}) = \log p(\mathbf{f}|\mathcal{D})$, assumed to be Gaussian, reaches maximum with $\mathbf{g}_\psi(\mathbf{f}) = \mathbf{0}$. However, we note that in the equation above, $\mathbf{m}_{f|D}$ as a value of $\mathbf{f}$ is expressed in terms of both $\mathbf{w}$ and $\mathbf{W}$, which in turn are functions of $\mathbf{f}$ as given in Eqs. (8.32) and (8.33), i.e., $\mathbf{m}_{f|D}$ given above is not in a closed form, as $\mathbf{f}$ and parameters $\mathbf{w}$ and $\mathbf{W}$ are interdependent. We instead need to carry out an iteration during which both $\mathbf{f}$ and the parameters $\mathbf{w}$ and $\mathbf{W}$ are updated alternatively:

$$\begin{aligned}\mathbf{f}_{n+1} &= \mathbf{f}_n - \mathbf{H}_\psi^{-1}\,\mathbf{g}_\psi = \mathbf{f}_n + (\mathbf{K}^{-1} - \mathbf{W})^{-1}\left(\mathbf{w} - \mathbf{K}^{-1}\mathbf{f}_n\right) \\ &= \mathbf{f}_n + (\mathbf{K}^{-1} - \mathbf{W})^{-1}\left[-(\mathbf{K}^{-1} - \mathbf{W})\mathbf{f}_n + \mathbf{w} - \mathbf{W}\mathbf{f}_n\right] \\ &= (\mathbf{K}^{-1} - \mathbf{W})^{-1}\left(\mathbf{w} - \mathbf{W}\mathbf{f}_n\right)\end{aligned} \tag{8.37}$$

This iterative process converges to $\mathbf{f} = \mathbf{m}_{f|D}$, at which $p(\mathbf{f}|\mathcal{D})$ is maximized, and

$$\mathbf{g}_\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\psi(\mathbf{f}) = \mathbf{w} - \mathbf{K}^{-1}\mathbf{f} = \mathbf{0}, \qquad \text{i.e.,} \qquad \mathbf{f} = \mathbf{m}_{f|D} = \mathbf{K}\,\mathbf{w} \tag{8.38}$$

Now that $\mathbf{w}$ and $\mathbf{W}$ as well as $\mathbf{f} = \mathbf{m}_{f|D}$ are available, we can further obtain $\mathbf{\Sigma}_{f|D}$ in Eq. (8.34), and the posterior $p(\mathbf{f}|\mathcal{D}) \approx \mathcal{N}(\mathbf{m}_{f|D}, \mathbf{\Sigma}_{f|D})$.

Having found $p(\mathbf{f}|\mathcal{D})$, we can proceed to carry out classification of any test set $\mathbf{X}_*$ based on the probability (similar to Eq. (8.16)):

$$p(\mathbf{y}_* = 1|\mathbf{X}_*, \mathcal{D}) = E_f[\sigma(\mathbf{f}(\mathbf{X}_*))] = \int \sigma(\mathbf{f}(\mathbf{X}_*))\, p(\mathbf{f}|\mathbf{X}_*, \mathcal{D})\, d\mathbf{f} \tag{8.39}$$

We first approximate the posterior of $\mathbf{f}_* = \mathbf{f}(\mathbf{X}_*)$ in the equation as a Gaussian:

$$p(\mathbf{f}|\mathbf{X}_*, \mathcal{D}) \approx \mathcal{N}(\mathbf{m}_{f_*}, \mathbf{\Sigma}_{f_*}) \tag{8.40}$$

where the mean $\mathbf{m}_{f_*}$ and covariance $\mathbf{\Sigma}_{f_*}$ can be obtained based on the fact that both $\mathbf{f}_* = \mathbf{f}(\mathbf{X}_*)$ and $\mathbf{f} = \mathbf{f}(\mathbf{X})$ are the same Gaussian process, i.e., their joint probability $p(\mathbf{f}, \mathbf{f}_*)$ is a Gaussian. The method is therefore the same as what is discussed in the method of GPR. Specifically, we take the following steps:

- Find the mean $\mathbf{m}_{f_*|f} = E(\mathbf{f}_*|\mathbf{f})$ and covariance $\mathbf{\Sigma}_{f_*|f} = \mathrm{Cov}[\mathbf{f}_*|\mathbf{f}]$ of $p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{f})$ conditioned on the latent function $\mathbf{f}$, same as in Eq. (8.8) for GPR:

$$\begin{cases} \mathbf{m}_{f_*|f} = \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{f} \\ \mathbf{\Sigma}_{f_*|f} = \mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{K}_* \end{cases} \tag{8.41}$$

- Find $\mathbf{m}_{f_*} = E_f(\mathbf{f}_*)$ as the expectation of $\mathbf{m}_{f_*|f} = \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{f}$ conditioned on $\mathbf{f}$, i.e. find the average of $\mathbf{m}_{f_*|f}$ over $\mathbf{f}$ based on $p(\mathbf{f}|\mathcal{D})$ (marginalize over $\mathbf{f}$):

$$\begin{aligned}\mathbf{m}_{f_*} &= E_f(\mathbf{m}_{f_*|f}) = E_f\left(\mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{f}\right) = \mathbf{K}_*^T\mathbf{K}^{-1}E_f(\mathbf{f}) = \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{m}_f \\ &= \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{K}\,\mathbf{w}_{\mathbf{m}_f} = \mathbf{K}_*^T\mathbf{w}_{\mathbf{m}_f}\end{aligned} \tag{8.42}$$

where $\mathbf{m}_f = E_f(\mathbf{f}) = \mathbf{K}\mathbf{w}_{m_f}$ given in Eq. (8.38).

- Find the covariance $\boldsymbol{\Sigma}_{m_{f_*|f}} = E_f[(\mathbf{m}_{f_*|f} - \mathbf{m}_{f_*})^2]$. Given the covariance $\boldsymbol{\Sigma}_{f|D} = (\mathbf{K}^{-1} - \mathbf{W})^{-1}$ of $\mathbf{f}$ in Eq. (8.34), we can further find the covariance of $\mathbf{m}_{f_*|f} = \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{f}$ as a linear combination of $\mathbf{f}$ (Recall if $\mathbf{y} = \mathbf{A}\mathbf{x}$, then $\boldsymbol{\Sigma}_y = \mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T$):

$$\boldsymbol{\Sigma}_{m_{f_*|f}} = \mathbf{K}_*^T\mathbf{K}^{-1}\,\boldsymbol{\Sigma}_{f|D}\,\mathbf{K}^{-1}\mathbf{K}_* = \mathbf{K}_*^T\mathbf{K}^{-1}(\mathbf{K}^{-1} - \mathbf{W})^{-1}\mathbf{K}^{-1}\mathbf{K}_* \quad (8.43)$$

- Find the covariance $\boldsymbol{\Sigma}_{f_*}$ of $\mathbf{f}_*$ as the sum of $\boldsymbol{\Sigma}_{f_*|f} = E[(\mathbf{f}_* - \mathbf{m}_{f_*|f})(\mathbf{f}_* - \mathbf{m}_{f_*|f})^T]$ for the variation of $\mathbf{f}_*$ with respect to $\mathbf{m}_{f_*|f}$, and $\boldsymbol{\Sigma}_{m_{f_*|f}} = E_f[(\mathbf{m}_{f_*|f} - \mathbf{m}_{f_*})^2]$ for the variation of $\mathbf{m}_{f_*|f}$ with respect to $\mathbf{m}_{f_*}$:

$$\begin{aligned}
\boldsymbol{\Sigma}_{f_*} &= \boldsymbol{\Sigma}_{f_*|f} + \boldsymbol{\Sigma}_{m_{f_*|f}} \\
&= (\mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{K}_*) + (\mathbf{K}_*^T\mathbf{K}^{-1}(\mathbf{K}^{-1} - \mathbf{W})^{-1}\mathbf{K}^{-1}\mathbf{K}_*) \\
&= \mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{K}_* + \mathbf{K}_*^T\mathbf{K}^{-1}[\mathbf{K} - \mathbf{K}(\mathbf{K} - \mathbf{W}^{-1})^{-1}\mathbf{K}]\mathbf{K}^{-1}\mathbf{K}_* \\
&= \mathbf{K}_{**} - \mathbf{K}_*^T(\mathbf{K} - \mathbf{W}^{-1})^{-1}\mathbf{K}_* \qquad (8.44)
\end{aligned}$$

Here we have used the identity for $(\mathbf{A} + \mathbf{B})^{-1}$ given in Section A.2.4.

Now that the posterior $p(\mathbf{f}|\mathbf{X}_*, \mathcal{D}) = \mathcal{N}(\mathbf{m}_{f_*}, \boldsymbol{\Sigma}_{f_*})$ is approximated as a Gaussian with mean and covariance given in Eqs. (8.42) and (8.44), we can finally carry out Eq. (8.16) to find the probability for the test points in $\mathbf{X}_*$ to belong to class C_1:

$$\begin{aligned}
p(\mathbf{y}_* = 1|\mathbf{X}_*, \mathcal{D}) &= \int \sigma(\mathbf{f}_*)\, p(\mathbf{f}_*|\mathbf{X}_*, \mathcal{D})\, d\mathbf{f}_* \\
&= E_f[\sigma(\mathbf{f}(\mathbf{X}_*))] = \sigma(E_f(\mathbf{f}(\mathbf{X}_*))) = \sigma(\mathbf{m}_{f_*}) \qquad (8.45)
\end{aligned}$$

Same as in GPR, the certainty or confidence of this classification result is represented by the variances on the diagonal of the covariance $\boldsymbol{\Sigma}_{f_*}$.

The Matlab code for the essential parts of this algorithm is listed below. Here `X` and `y` are for the training data $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$, and `Xs` is an array composed of test vectors. First, the essential segment of the main program listed below takes in the training and test data, generates the covariance matrices $\mathbf{K}$, $\mathbf{K}_*$, and $\mathbf{K}_{**}$ represented by `K`, `Ks`, and `Kss`, respectively. The function `Kernel` is exactly the same as the one used for Gaussian process regression. This code segment then further calls a function `findPosteriorMean` which finds the mean and covariance of $p(\mathbf{f}|\mathbf{X}, \mathbf{y})$ based on covariance of training data ($\mathbf{K} = \text{Cov}[\mathbf{X}]$ and $\mathbf{y}$, and computes the mean $\mathbf{m}_{f_*|y}$ and covariance $\boldsymbol{\Sigma}_{f_*|y}$ based on Eqs. (8.42) and (8.44), respectively. The sign function of $\mathbf{m}_{f_*|y}$ indicates the classification of the test data points in $\mathbf{X}_*$.

```
K=Kernel(X,X);                % cov(f,f), covariance of prior p(f|X)
Ks=Kernel(X,Xs);              % cov(f_*,f)
Kss=Kernel(Xs,Xs);            % cov(f_*,f_*)
[Sigmaf W w]=findPosteriorMean(K,y);
                              % get mean/covariance of p(f|D), W, w
```

```
    meanfD=Ks'*w;                % mean of p(f_*|X_*,D)
    SigmafD=Kss-Ks'*inv(K-inv(W))*Ks;  % covariance of p(f_*|X_*,D)
    ys=sign(meanfD);             % binary classification of test data
    p=1./(1+exp(-meanfD));       % p(y_*=1|X_*,D) as logistic function
```

The function `findPosteriorMean` listed below uses Newton's method to find the mean and covariance of the posterior $p(\mathbf{f}|\mathcal{D})$ of the latent function $\mathbf{f}$ based on the training data, and returns them in `meanf` and `covf`, respectively, together with `w` and `W` for the gradient and Hessian of the likelihood $p(\mathbf{y}|\mathbf{f})$, to be used for computing $\mathbf{m}_{f_*|D}$ and $\mathbf{\Sigma}_{f_*|D}$.

```
function [meanf covf W w]=findPosteriorMean(K,y)
    % K: covariance of prior of p(f|X)
    % y: labeling of training data X
    % w: gradient vector of log p(y|f)
    % W: Hessian matrix of log p(y|f)
    % meanf: mean of p(f|X,y)
    % covf: covariance of p(f|X,y)
    n=length(y);        % number of training samples
    f0=zeros(n,1);      % initial value of latent function
    f=f0;
    er=1;
    while er > 10^(-9)  % Newton method to get f that maximizes p(f|X,y)
        e=exp(-y.*f);
        w=y.*e./(1+e);            % update w
        W=diag(-e./(1+e).^2);     % update W
        f=inv(inv(K)-W)*(w-W*f0); % iteration to get f from previous f0
        er=norm(f-f0);            % difference between 2 consecutive f's
        f0=f;                     % update f
    end
    meanf=f;                      % mean of f
    covf=inv(inv(K)-W);           % coviance of f
end
```

Example 8.2 The GPC method is trained by the two classes shown in Fig. 8.5, represented by 100 red points and 80 blue points drawn from two Gaussian distributions $\mathcal{N}(\mathbf{m}_0, \mathbf{\Sigma}_0)$ and $\mathcal{N}(\mathbf{m}_1, \mathbf{\Sigma}_1)$, where

$$\mathbf{m}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{m}_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \mathbf{\Sigma}_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{\Sigma}_1 = \begin{bmatrix} 1 & 0.9 \\ 0.9 & 1 \end{bmatrix} \tag{8.46}$$

The 3-D plot of these two normalized Gaussian distributions is also shown in the figure.

The three classification results shown in the three panels in Fig. 8.6 correspond to three different values $\alpha = 0.4,\ 0.2,\ 0.03$ for the kernel function. On the left, the 2-D space is partitioned into red and blue regions corresponding to the two

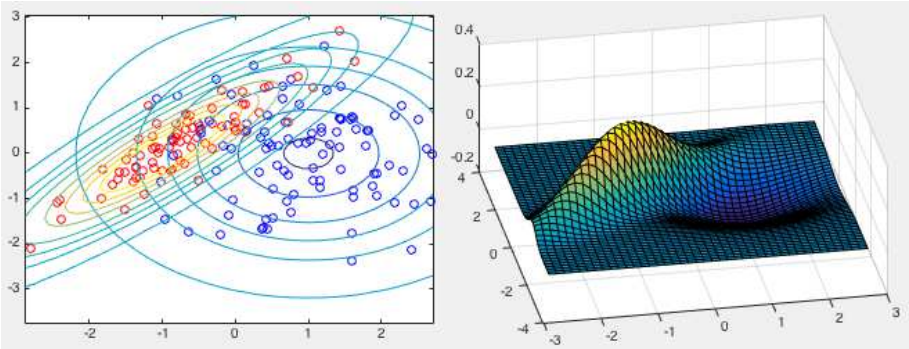


Figure 8.5 Gaussian Process for Classification (Binary)

classes based on the sign function of $\mathbf{m}_{f_*|y}$; on the right, the 3-D distributions of $\sigma(\mathbf{m}_{f_*|y})$ represent the estimated probability for $\mathbf{x}_*$ to belong to either class (not normalized), to be compared with the original Gaussian distributions in Fig. 8.5, from which the training samples were drawn.

We see that wherever there is evidence represented by the training samples of either class in red or blue, there are high probabilities for the neighboring points to belong to either C_1 or C_0 represented by the positive or negative peaks in the 3-D plots. Data points far away from any evidence will have low probability to belong to either class.

We make the following observations for three different values of the parameter α in SE:

- $\alpha = 0.4$ (top row), the space is partitioned into three regions, with 20 out of 180 training points misclassified. The estimated distribution is smooth.
- $\alpha = 0.2$ (middle row), the space is fragmented into several more pieces for the two classes (blue islands inside the red region and vice versa), with 5 out of the 180 training points misclassified. The estimated distribution is jagged.
- $\alpha = 0.03$ (bottom row), the space is partitioned into still more pieces, with all 180 training points correctly classified. The estimated distribution is spiky.

Although the error rate is lowest when α is small, the classification result is not necessarily the best as it may well be an overfitting of the noisy data. We conclude that by adjusting parameter α, we can make proper tradeoff between error rate and overfitting.

8.3 Gaussian Process Classifier – Multi-Class

The binary GPC considered previously can be generalized to multi-class GPC based on softmax function, similar to the how binary classification based on

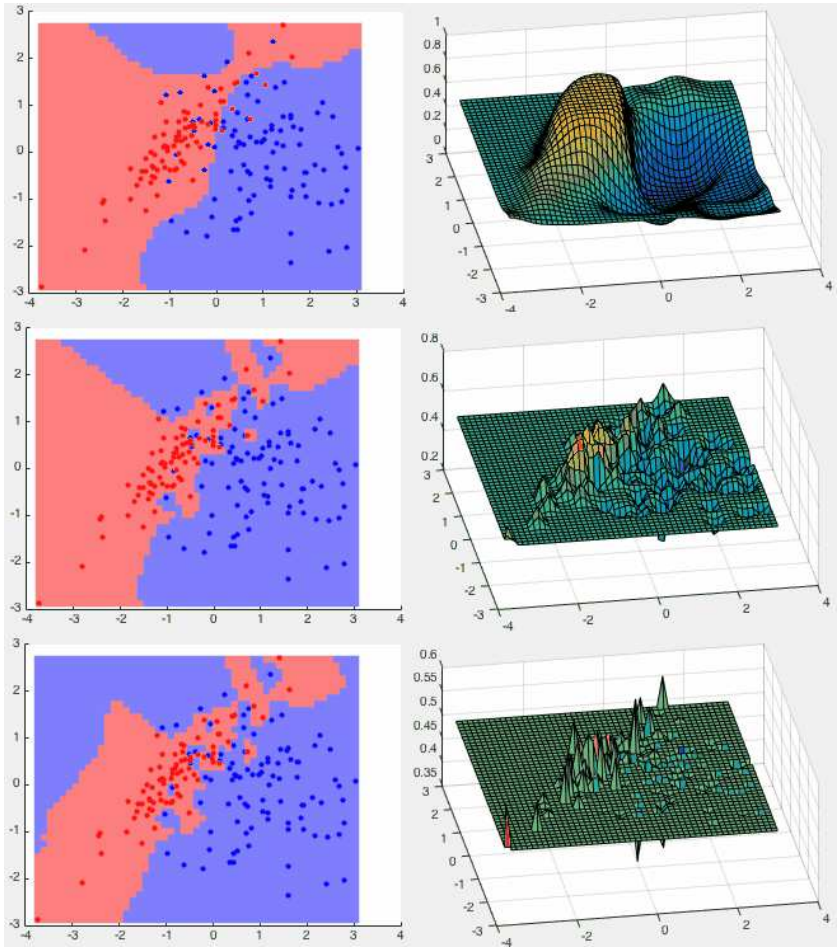


Figure 8.6 Classification Results with Different α Values

logistic function is generalized to multi-class classification. First of all, we define the following variables for each class C_k, $(k = 1, \cdots, K)$ of the K classes $\{C_1, \cdots, C_K\}$:

- **Class labeling:**

  Binary labeling $\mathbf{y}_k = [y_{1k}, \cdots, y_{Nk}]^T$ indicating whether each of the N samples in the training set $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ belongs to C_k:

$$y_{nk} = \begin{cases} 1 & \text{if } \mathbf{x}_n \in C_k \\ 0 & \text{if } \mathbf{x}_n \notin C_k \end{cases} \tag{8.47}$$

  In other words, each training sample $\mathbf{x}_n \in C_k$ is labeled by K binary variables $y_{nk} = 1$ and $y_{nl} = 0$ for all $l = 1, \cdots, K$, $l \neq k$ (same as in softmax regression), as well as an integer labeling $y_n = k$.

- **Latent functions:**

  Latent function $\mathbf{f}_k = [f_{1k}, \cdots, f_{Nk}]^T$, of which the nth component $f_{nk} = f_k(\mathbf{x}_n)$ $(n = 1, \cdots, N)$ is kth Gaussian process associated with C_k evaluated at the nth training sample $\mathbf{x}_n$ in the training set $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$.

- **Probability modeling:**

  Probability $\mathbf{p}_k = [p_{1k}, \cdots, p_{Nk}]^T$, of which the nth component p_{nk} $(n = 1, \cdots, N)$ is the probability for $\mathbf{x}_n \in C_k$, modeled by the Softmax function based on f_{nk} (same as in Eq. (7.31) in softmax regression):

$$p_{nk} = p(\hat{y} = k|\mathbf{x}_n) = \frac{e^{f_{nk}}}{\sum_{l=1}^{K} e^{f_{nl}}}, \qquad (k = 1, \cdots, K, \;\; n = 1, \cdots, N) \tag{8.48}$$

  Note that if $y_{nk} = 1$ then necessarily $y_{nl} = 0$ for all $l = 1, \cdots, K$ but $l \neq k$, i.e., the K variables $y_{n1}, \cdots, y_{nK}$ are not independent. given $p(y_{nk} = 1)$, we do not need to consider $p(y_{nl} = 0)$ for any $l \neq k$.

The probability for $\mathbf{x}_n$ to be correctly classified into class C_{y_n} can be written as the following product of K factors (of which $K - 1$ are equal to 1, same as Eq. (7.32) in softmax regression):

$$p(\hat{y} = y_n|\mathbf{x}_n, \mathbf{f}) = \prod_{k=1}^{K} p(y_{nk} = 1|\mathbf{x}_n, \mathbf{f})^{y_{nk}} = \prod_{k=1}^{K} p_{nk}^{y_{nk}} \tag{8.49}$$

Based on $\mathbf{y}_k$, $\mathbf{f}_k$, and $\mathbf{p}_k$ for all $k = 1, \cdots, K$, we further define the following KN dimensional vectors:

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_K \end{bmatrix} = \begin{bmatrix} y_{11} \\ \vdots \\ y_{N1} \\ \vdots \\ y_{1K} \\ \vdots \\ y_{NK} \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_K \end{bmatrix} = \begin{bmatrix} f_{11} \\ \vdots \\ f_{N1} \\ \vdots \\ f_{1K} \\ \vdots \\ f_{NK} \end{bmatrix}, \quad \mathbf{p} = \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_K \end{bmatrix} = \begin{bmatrix} p_{11} \\ \vdots \\ p_{N1} \\ \vdots \\ p_{1K} \\ \vdots \\ p_{NK} \end{bmatrix} \tag{8.50}$$

The posterior of $\mathbf{f}$ given the training set $\mathcal{D}$ can be found based on the Bayesian theorem as (same as in Eq. (8.17) in the binary case):

$$p(\mathbf{f}|\mathcal{D}) = p(\mathbf{f}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{f})\; p(\mathbf{f}|\mathbf{X})}{p(\mathbf{y}|\mathbf{X})} \propto p(\mathbf{y}|\mathbf{f})\; p(\mathbf{f}|\mathbf{X}) \tag{8.51}$$

We first find the likelihood based on all p_{nk} (same as Eq. (7.33) in softmax regression):

$$L(\mathbf{f}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{f}) = \prod_{n=1}^{N} p(\hat{y} = y_n|\mathbf{x}_n, \mathbf{f}) = \prod_{n=1}^{N} \left(\prod_{k=1}^{K} (p_{nk})^{y_{nk}} \right) = \prod_{n=1}^{N} \prod_{k=1}^{K} \left(\frac{e^{f_{nk}}}{\sum_{h=1}^{K} e^{f_{nh}}} \right)^{y_{nk}} \tag{8.52}$$

As each sample $\mathbf{x}_n$ belongs to only one of the K classes, only one of $\{y_{n1}, \cdots, y_{nK}\}$ can be 1 while all others are 0, consequently, the product above contains only N probabilities raised to the power of $y_{nk} = 1$, each for one of the N samples belonging to a certain class, while all other probabilities raised to the power of $y_{nk} = 0$ become 1 and not considered.

We next assume the prior probability of the latent function $\mathbf{f}_k$ for each class C_k to be a zero-mean Gaussian process $p(\mathbf{f}_k|\mathbf{X}) = \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_k)$, where the covariance matrix $\mathbf{\Sigma}_k$ is constructed based on the squared exponential (SE) for its mn-th component:

$$\begin{aligned}\mathrm{Cov}[f_k(\mathbf{x}_m), f_k(\mathbf{x}_n)] = \mathrm{Cov}[f_{mk}, f_{nk}] &= k(\mathbf{x}_m, \mathbf{x}_n) \\ &= \exp\left(-\frac{1}{a^2}||\mathbf{x}_m - \mathbf{x}_n||^2\right), \qquad (m, n = 1, \cdots, N)\end{aligned} \tag{8.53}$$

Then the prior of $\mathbf{f}$ containing all K such latent functions is also a Gaussian $p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{0}, \mathbf{K})$, where $\mathbf{0}$ is a KN-dimensional zero vector and $\mathbf{K}$ is a $KN \times KN$ block diagonal matrix

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{K}_2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \mathbf{0} & \mathbf{K}_K \end{bmatrix} \tag{8.54}$$

with $\mathbf{K}_1 = \cdots = \mathbf{K}_K$ as defined in Eq. (8.4) on the diagonal. All off-diagonal blocks are zero as the latent functions of different classes are uncorrelated.

Having found both the likelihood $p(\mathbf{y}|\mathbf{f})$ and prior $p(\mathbf{f}|\mathbf{X})$, we can write the posterior as

$$p(\mathbf{f}|\mathcal{D}) \propto p(\mathbf{f}|\mathbf{X}, \mathbf{y}) \propto p(\mathbf{y}|\mathbf{f})\, p(\mathbf{f}|\mathbf{X}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \left(\frac{e^{f_{nk}}}{\sum_{h=1}^{K} e^{f_{nh}}}\right)^{y_{nk}} \mathcal{N}(\mathbf{0},\, \mathbf{K}) \tag{8.55}$$

We note that the likelihood $p(\mathbf{y}|\mathbf{f})$ is not Gaussian, as $y_{nk} \in \{0, 1\}$ is a binary labeling, instead of a continuous variable. As a product of the Gaussian prior and non-Gaussian likelihood, the posterior $p(\mathbf{f}|\mathcal{D})$ is not a Gaussian process either. However, for convenience, we still assume it is approximately Gaussian $p(\mathbf{f}|\mathcal{D}) \approx \mathcal{N}(\mathbf{f},\, \mathbf{m}_{f|D},\, \mathbf{\Sigma}_{f|D})$, of which the mean $\mathbf{m}_{f|D}$ and covariance $\mathbf{\Sigma}_{f|D}$ are to be found.

Taking log of the posterior, we get

$$\begin{aligned}\psi(\mathbf{f}) = \log p(\mathbf{f}|\mathcal{D}) &= \log p(\mathbf{y}|\mathbf{f}) + \log p(\mathbf{f}|\mathbf{X}) \\ &= \sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} \left(f_{nk} - \log \sum_{h=1}^{K} e^{f_{nh}}\right) + \log \mathcal{N}(\mathbf{0}, \mathbf{K}) \\ &= \mathbf{y}^T \mathbf{f} - \sum_{n=1}^{N} \log \sum_{h=1}^{K} e^{f_{nh}} - \frac{NK}{2} \log(2\pi) - \frac{1}{2} \log |\mathbf{K}| - \frac{1}{2} \mathbf{f}^T \mathbf{K}^{-1} \mathbf{f}\end{aligned} \tag{8.56}$$

where we have used the fact that $\sum_{k=1}^{K} y_{nk} = 1$. The gradient of $\psi(\mathbf{f})$ is

$$\begin{aligned}\mathbf{g}_\psi = \frac{d}{d\mathbf{f}}\psi(\mathbf{f}) &= \frac{d}{d\mathbf{f}}\left(\mathbf{y}^T\mathbf{f} - \sum_{n=1}^{N}\log\sum_{h=1}^{K} e^{f_{nh}} - \frac{NK}{2}\log(2\pi) - \frac{1}{2}\log|\mathbf{K}| - \frac{1}{2}\mathbf{f}^T\mathbf{K}^{-1}\mathbf{f}\right) \\ &= \mathbf{y} - \mathbf{p} - \mathbf{K}^{-1}\mathbf{f} \end{aligned} \tag{8.57}$$

where the second term $\mathbf{p}$ comes from

$$\sum_{n=1}^{N}\frac{d}{d\mathbf{f}}\log\sum_{h=1}^{K} e^{f_{nh}} = \sum_{n=1}^{N}\begin{bmatrix} 0 \\ \vdots \\ 0 \\ \frac{e^{f_{nk}}}{\sum_{h=1}^{K} e^{f_{nh}}} \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \sum_{n=1}^{N}\begin{bmatrix} 0 \\ \vdots \\ 0 \\ p_{nk} \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{p} \tag{8.58}$$

Note that $\mathbf{p} = \mathbf{p}(\mathbf{f})$ is a function of $\mathbf{f}$.

As the posterior $p(\mathbf{f}|\mathcal{D}) \approx \mathcal{N}(\mathbf{m}_{f|D}, \mathbf{\Sigma}_{f|D})$ is approximated as a Gaussian, which reaches maximum at its mean $\mathbf{m}_{f|D}$, and so does the log prior $\psi(\mathbf{f}) = \log p(\mathbf{f}|\mathcal{D})$, and the gradient of $psi(\mathbf{f})$ is zero at $\mathbf{f} = \mathbf{m}_{f|D}$:

$$\mathbf{g}_\psi(\mathbf{f})\Big|_{\mathbf{f}=\mathbf{m}_{f|D}} = \mathbf{y} - \mathbf{p} - \mathbf{K}^{-1}\mathbf{f}\Big|_{\mathbf{f}=\mathbf{m}_{f|D}} = \mathbf{y} - \mathbf{p}_{m_f} - \mathbf{K}^{-1}\mathbf{m}_{f|D} = \mathbf{0}, \tag{8.59}$$

i.e.,

$$\mathbf{m}_{f|D} = \mathbf{K}(\mathbf{y} - \mathbf{p}_{m_f}) \tag{8.60}$$

where $\mathbf{p}_{m_f} = \mathbf{p}(\mathbf{m}_{f|D})$ is the vector defined above evaluated at $\mathbf{f} = \mathbf{m}_{f|D}$.

We further get the Hessian matrix of $\psi(\mathbf{f})$:

$$\mathbf{H}_\psi(\mathbf{f}) = \frac{d^2}{d\mathbf{f}^2}\psi(\mathbf{f}) = \frac{d}{d\mathbf{f}}\mathbf{g}_\psi = \frac{d}{d\mathbf{f}}\left(\mathbf{y} - \mathbf{K}^{-1}\mathbf{f} - \mathbf{p}\right) = -\mathbf{K}^{-1} - \mathbf{W} = -\mathbf{\Sigma}_{f|D}^{-1} \tag{8.61}$$

The last equality is due to the property of the Gaussian distribution given in Section B.1.5, from which we get the KN by KN covariance matrix of $p(\mathbf{f}|\mathcal{D})$:

$$\mathbf{\Sigma}_{f|D} = -\mathbf{H}_\psi^{-1}(\mathbf{f}) = (\mathbf{K}^{-1} + \mathbf{W})^{-1} \tag{8.62}$$

Here $\mathbf{W} = d\mathbf{p}/d\mathbf{f}$ is a $KN \times KN$ Jacobian matrix of $\mathbf{p}$, of which the $ijkl$-th component $(i, j = 1, \cdots, N,\ \ k, l = 1, \cdots, K)$ is:

$$\begin{aligned}\frac{\partial}{\partial f_{jl}}p_{ik} = \frac{\partial}{\partial f_j^l}\left[\frac{e^{f_{ik}}}{\sum_{h=1}^{K} e^{f_{ih}}}\right] &= \frac{e^{f_{ik}}\left(\sum_{h=1}^{K} e^{f_{ih}}\right)\delta_{kl} - e^{f_{ik}}e^{f_{jl}}}{\left(\sum_{h=1}^{K} e^{f_{ih}}\right)^2}\delta_{ij} \\ &= \left[\frac{e^{f_{ik}}}{\sum_{h=1}^{K} e^{f_{ih}}}\delta_{kl} - \frac{e^{f_{ik}}e^{f_{jl}}}{\left(\sum_{h=1}^{K} e^{f_{ih}}\right)^2}\right]\delta_{ij} = (p_{ik}\delta_{kl} - p_{ik}p_{jl})\delta_{ij} \end{aligned} \tag{8.63}$$

and $\mathbf{W}$ can be written also in two terms:

$$\mathbf{W} = \text{diag}(\mathbf{p}) - \mathbf{P}\mathbf{P}^T \tag{8.64}$$

where the first term $\text{diag}(\mathbf{p})$ is a diagonal matrix containing all KN components of $\mathbf{p}$ along the diagonal ($k = l$ and $i = j$), and $\mathbf{P}$ in the second term is a KN by N matrix composed of K $N \times N$ diagonal matrices $\text{diag}(\mathbf{p}_k)$:

$$\mathbf{P} = \begin{bmatrix} \text{diag}(\mathbf{p}_1) \\ \vdots \\ \text{diag}(\mathbf{p}_K) \end{bmatrix}, \quad \text{diag}(\mathbf{p}_k) = \begin{bmatrix} p_{1k} & 0 & \cdots & 0 \\ 0 & p_{2k} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & p_{Nk} \end{bmatrix}, \quad (k = 1, \cdots, K) \tag{8.65}$$

so that

$$\begin{aligned} \mathbf{P}\mathbf{P}^T &= \begin{bmatrix} \text{diag}(\mathbf{p}_1) \\ \vdots \\ \text{diag}(\mathbf{p}_K) \end{bmatrix} [\text{diag}(\mathbf{p}_1), \cdots, \text{diag}(\mathbf{p}_K)] \\ &= \begin{bmatrix} \text{diag}(\mathbf{p}_1^2) & \cdots & \text{diag}(\mathbf{p}_1)\,\text{diag}(\mathbf{p}_K) \\ \vdots & \ddots & \vdots \\ \text{diag}(\mathbf{p}_K)\,\text{diag}(\mathbf{p}_1) & \cdots & \text{diag}(\mathbf{p}_K^2) \end{bmatrix} \end{aligned} \tag{8.66}$$

with

$$\text{diag}(\mathbf{p}_k)\,\text{diag}(\mathbf{p}_l) = \begin{bmatrix} p_1^k p_1^l & 0 & \cdots & 0 \\ 0 & p_1^k p_1^l & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & p_{Nk} p_{Nl} \end{bmatrix} \tag{8.67}$$

Having found both the gradient $\mathbf{g}_\psi = \mathbf{y} - \mathbf{p} - \mathbf{K}^{-1}\mathbf{f}$ and Hessian $\mathbf{H}_\psi = -(\mathbf{K}^{-1} + \mathbf{W})$, we can further find the mean $\mathbf{m}_{f|D}$ at which $\psi(\mathbf{f})$ achieves maximum by the following iteration of Newton's method:

$$\begin{aligned} \mathbf{f}_{n+1} &= \mathbf{f}_n - \mathbf{H}_\psi^{-1}\mathbf{g}_\psi = \mathbf{f}_n + (\mathbf{K}^{-1} + \mathbf{W})^{-1}(\mathbf{y} - \mathbf{K}^{-1}\mathbf{f}_n - \mathbf{p}) \\ &= \mathbf{f}_n + (\mathbf{K}^{-1} + \mathbf{W})^{-1}(-(\mathbf{K}^{-1} + \mathbf{W})\mathbf{f}_n + \mathbf{W}\mathbf{f}_n + \mathbf{y} - \mathbf{p}) \\ &= (\mathbf{K}^{-1} + \mathbf{W})^{-1}(\mathbf{W}\mathbf{f}_n + \mathbf{y} - \mathbf{p}) \end{aligned} \tag{8.68}$$

We note that during the iteration, we need to update not only $\mathbf{f}$, but also $\mathbf{p}$ as a functions of $\mathbf{f}$ given in Eq. (8.58).

Now that we have got both $\mathbf{m}_{f|D}$ and $\mathbf{\Sigma}_{f|D}$ of $\mathbf{f}$, we can further get $\mathbf{m}_{f_*}$ and $\mathbf{\Sigma}_{f_*}$ of $\mathbf{f}_*$:

- Get mean $\mathbf{m}_{f_*}$:

  Similar to Eq. (8.42) in the binary case, we have the following based on $\mathbf{f} = \mathbf{K}(\mathbf{y} - \mathbf{p})$ in Eq. (8.60):

$$\begin{aligned} \mathbf{m}_{f_*} &= E_f(\mathbf{m}_{f_*|f}) = E_f(\mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{f}) = \mathbf{K}_*^T\mathbf{K}^{-1}E_f(\mathbf{f}) \\ &= \mathbf{K}_*^T\mathbf{K}^{-1}\left[\mathbf{K}(\mathbf{y} - \mathbf{p}_{m_f})\right] = \mathbf{K}_*^T(\mathbf{y} - \mathbf{p}_{m_f}) \end{aligned} \tag{8.69}$$

As the K classes are uncorrelated, i.e., $\mathbf{K}$ is block-diagonal, the above can be separated into K equations each for one of the classes:

$$\mathbf{m}_{f_*^k} = (\mathbf{K}_*^k)^T(\mathbf{y}^k - \mathbf{p}^k), \qquad (k = 1, \cdots, K) \tag{8.70}$$

- Get covariance $\mathbf{\Sigma}_{f_*}$:

  Similar to Eq. (8.44) in the binary case, we have the following based on $\mathbf{\Sigma}_{f|D} = (\mathbf{K}^{-1} + \mathbf{W})^{-1}$ in Eq. (8.62):

$$\mathbf{\Sigma}_{m_{f_*|f}} = \mathbf{K}_*^T\mathbf{K}^{-1}\,\mathbf{\Sigma}_{f|D}\,\mathbf{K}^{-1}\mathbf{K}_* = \mathbf{K}_*^T\mathbf{K}^{-1}(\mathbf{K}^{-1} + \mathbf{W})^{-1}\mathbf{K}^{-1}\mathbf{K}_* \tag{8.71}$$

  where $\mathbf{W} = \text{diag}(\mathbf{p}) - \mathbf{P}\mathbf{P}^T$ is given in Eq. (8.64) with $\mathbf{p}$ evaluated at $\mathbf{m}_{f|D}$. Then, similar to Eq. (8.44), we get

$$\mathbf{\Sigma}_{f_*} = \mathbf{K}_{**} - \mathbf{K}_*^T(\mathbf{K} + \mathbf{W}^{-1})^{-1}\mathbf{K}_* \tag{8.72}$$

Now we can further get the probability for $\mathbf{x}_*$ to belong to C_k based on the softmax function in Eq. (8.48)

$$p_*^k = \frac{e^{m_{f_*}^k}}{\sum_{l=1}^K e^{m_{f_*}^l}}, \qquad (k = 1, \cdots, K) \tag{8.73}$$

and classify $\mathbf{x}_*$ to class C_k if $p_*^k = \max\{p_*^1, \cdots, p_*^K\}$.

Moreover, the certainty or confidence of this classification result can be found from $\mathbf{\Sigma}_{f_*}$.

The Matlab code for the essential parts of the algorithm is listed below. First, the following code segment carries out the classification of n given test samples based on the N training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$.

```
K=Kernel(X,X);      % covariance of prior of p(f|X)
Ks=Kernel(X,Xs);
Kss=Kernel(Xs,Xs);
[meanf Sigmaf p W]=findPosteriorMeanMC(K,y,C);
                    % find mean and covariance of p(f|D), and W, p
Sigmafy=Kss-Ks'*inv(K+inv(W))*Ks; % covariance of p(f|D)
p=reshape(p,N,C);
y=reshape(y,N,C);
for k=1:C
    meanfD(:,k)=Ks'*(y(:,k)-p(:,k);
                    % mean of p(f_*|X_*,D) for kth class
end
for i=1:n           % for each of n test samples
    d=sum(exp(meanfD(i,:)));
                    % denominator of softmax function
    [pmax k]=max(exp(meanfD(i,:))/d);
                    % find class with max probability
    ys(i)=k;        % label ith sample as member of kth class
    pr(i,k)=pmax;   % probility of ith sample belonging to kth class
```

```
    end
```

The code segment above calls the following function which computes the mean and covariance of $p(\mathbf{f}|\mathbf{X},\mathbf{y})$ by Newton's method:

```
function [meanf Sigmaf p W]=findPosteriorMeanMC(K,y,C)
    % get mean and covariance of p(f|X,y) by Newton's method, given {X,y}
    n=length(y);          % number of training samples
    f0=zeros(n,1);        % initial value of latent function
    f=zeros(n,1);
    er=1;
    k=0;
    while er > 10^(-9)    % find f that maximizes p(f|X,y)
        k=k+1;
        [p W]=findp(f,N,C);
                          % call a function to find vector p and matrix W
        f=inv(inv(K)+W)*(W*f0+y-p);
                          % iteratively update value of f
        er=norm(f-f0);    % difference between consecutive iterations
        f0=f;             % update f
    end
    meanf=f;
    Sigmaf=inv(inv(K)+W);
end
```

The following function called by the previous function finds vector $\mathbf{p}$ and matrix $\mathbf{W}=\text{diag}(\mathbf{p})-\mathbf{P}\mathbf{P}^T$:

```
function [p W]=findp(f,N,C)  % find vector p and matrix W=diag(p)-P*P'
    F=reshape(f,N,C)';       % kth row contains N samples of class k
    p=zeros(C,N);            % initialize p
    for n=1:N                % for each of N training samples
        d=sum(exp(F(:,n)));  % sum of all C terms in denominator
        for k=1:C            % for all C classes
            p(k,n)=exp(F(k,n))/d;
        end
    end
    P=[];
    for k=1:C                % generate P
        P=[P; diag(p(k,:))]; % stack C diagonal matrices
    end
    p=reshape(p',N*C,1);     % convert p into a column vector
    W=diag(p)-P*P';          % generate W matrix
end
```

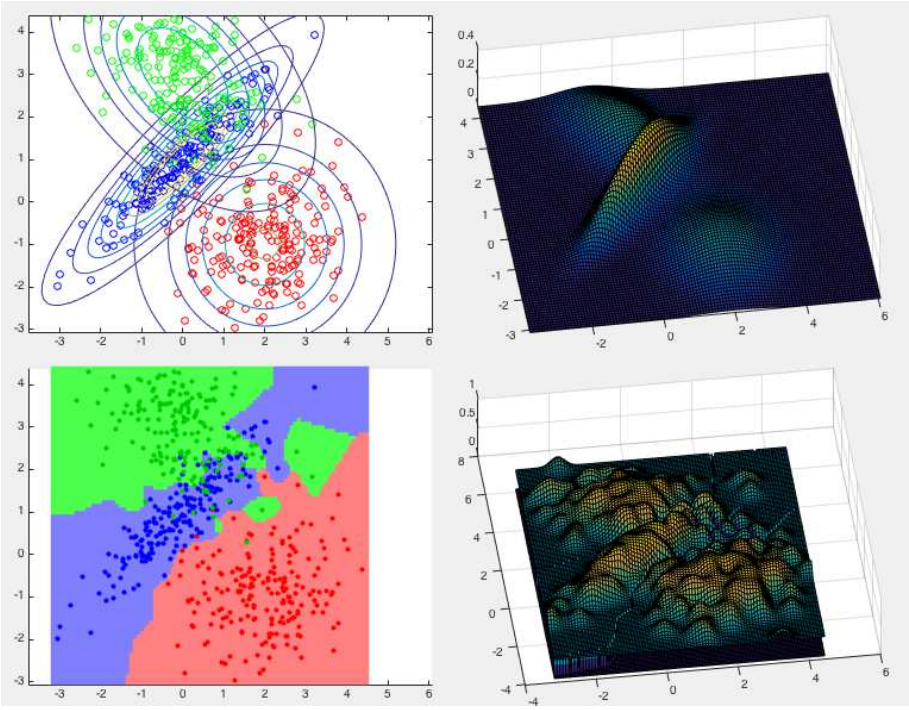


Figure 8.7 Gaussian Process for Classification (Multi-Class)

Example 8.3 This example illustrates the classification of the same dataset of three classes used before. The top two panels of Fig. 8.7 show the distributions of the three classes in the training data set, while the bottom two panels show the classification results in terms of the partitioning of the feature space (bottom left) and the posterior distribution $p(\mathbf{f}_*|\mathbf{X}, \mathbf{y}, \mathbf{X}_*)$ (bottom right), which can be compared with the distribution of the training set (top right). The confusion matrix of the classification result is shown below, with the error rate $30/600 = 0.045$:

$$\begin{bmatrix} 197 & 3 & 0 \\ 0 & 195 & 5 \\ 1 & 21 & 178 \end{bmatrix}$$

Example 8.4 This example illustrates the classification of the XOR data set as shown in Fig. 8.8. The confusion matrix of the classification result is shown below, with the error rate $13/400 = 0.0325$:

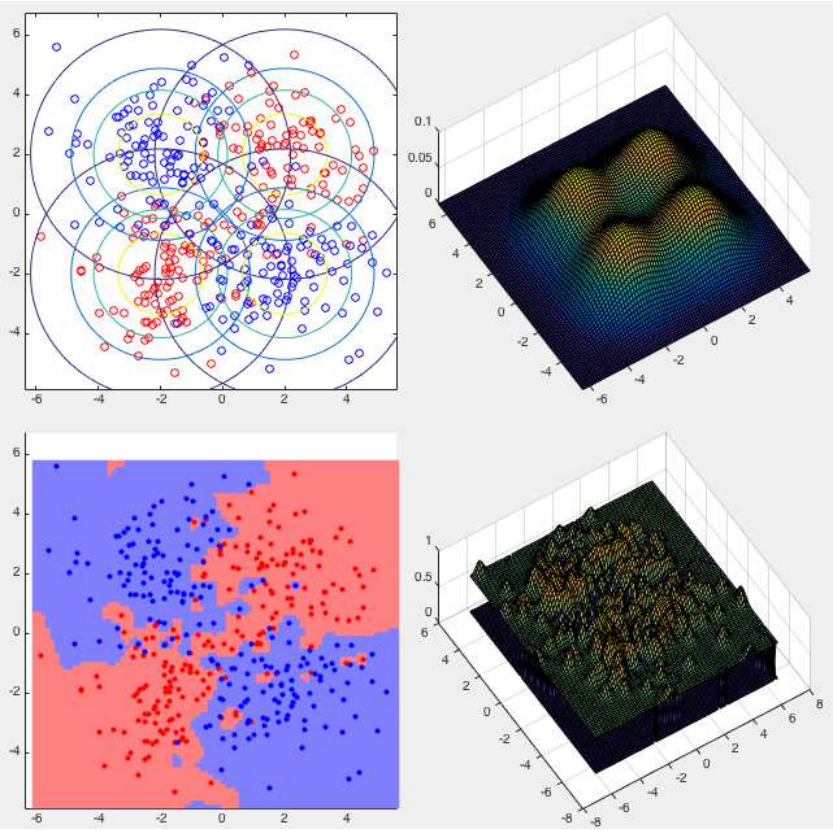


Figure 8.8 Gaussian Process Classification for the XOR Problem

$$\begin{bmatrix} 193 & 7 \\ 6 & 194 \end{bmatrix}$$

We see that in both examples, the error rates of the GPC method are lower than those of the naive Bayesian method. However, the naive Bayesian method does not have the overfitting problem, while in the method of GPC, we may need to carefully adjust the parameter of squared exponential for the kernel functions to make proper tradeoff between overfitting and error rate.

Problems

1. Implement the method of Gaussian process regression and carry out regression based on the dataset in problem as in Example 8.1 with $N = 8$ training samples randomly drawn from a noisy sinusoidal function $y = f(x) + n =$

$\cos(2\pi fx) + n$ in the domain `x=0:0.1:9`. Here $f = 0.2$ is the frequency and the noise has a Gaussian distribution with $(\mu_n = 0,\ \sigma_n^2)$. Try different values of σ (e.g., $\sigma = 0.2,\ 0.5,\ 1, \cdots$).

Plot your results in terms of the mean $E[f_*]$ for the regression function and variance $\pm\,\mathrm{Var}[f_*]$ for the range of certainty as three curves, the same as in Figs. 8.2 and 8.3.

Try different values for the hyperparameter α (variable a in Eq. (8.3) to see how the smoothness of the regression function is affected, and to make a proper tradeoff between underfitting and overfitting.

2. Implement the binary Gaussian process classifier and then carry out binary classification of the dataset below containing data points belonging to either of two classes (labeled by 1 or -1).

   /e176/programs/GPBCdata.txt

   Try three different α values for the squared exponential function to show underfitting, overfitting and proper fitting (similar Example 8.2).

3. Implement the Multi-Class Gaussian process classifier and then carry out multi-class classification of the dataset below: containing data points belonging to any of three classes (labeled by 1, 2 or 3).

   /e176/programs/GPMCdata.txt

   Try three different α values for the squared exponential function to show underfitting, overfitting and proper fitting.

Part III

Feature Extraction

In Part III, we will discuss various methods for *feature selection* or *extraction*, which can be considered as a preprocessing stage for the task of *pattern classification*, one of the main tasks in machine learning to be discussed in Part IV, for the goal of classifying a set of *patterns* into one of some K classes or clusters. Here the generic term *pattern* meanss any objects of interest each described by a set of d *features* denoted as a column vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ in a d-dimensional *feature space*, of which each component is some value that quantitatively measures the corresponding feature used to describe the pattern. The task of classification can therefore be considered as to partition the feature space into K regions each corresponding to one of the K classes or clusters.

Obviously the computational complexity of an operation on the dataset is a function (e.g., linear, polynomial, expential, etc.) of the dimensionality d of the feature space. When d is large, it is often desirable to reduce it to some significantly smaller value $d' \ll d$, while still maintaining most of the information in the dataset relevant to the task at hand, such as the variation of the features that describe the patterns or distinguish different classes of the patterns. For example, in machine learning and data science, it is often desirable to visualize the dataset in a high ($d > 3$) dimensional featue space to gain some intuitive sense of the dataset. It is therefore necessary to reduce the dimensionality to three or even two while maintaining the signal variation of interest in the data as much as possible. As another example in classification, the computational complexity of an algorithm may be too high if the dimensionality of the feature space is high. To reduce the computational complexity, we may consider reducing the dimensionality while still preserving the *separability* of the data points of different classes.

Feature selection or feature extraction can be used to reduce the dimensionality of a given dataset, by either selecting d' features directly from the d original ones (with $C_d^{d'} = d!/d'!(d-d')!$ ways to do so), or generating d' new features each as a function, such as the linear combinations, of the d original ones. In either case, the essential information of interest in the data needs to be maximally maintained in the resulting lower dimensional space spanned by the d' new features.

In the following chapters we will first consider how information of interest, such as class separability, can be measured quantitatively, and then discuss specific methods for information extraction and dimensionality reduction, such as the methods of *principal component analysis (PCA)*, *independent component analysis (ICA)* and their variations, widely used in signal processing, data science and machine learning.

9 Feature Selection

9.1 Distances and Separability Measurements

Before considering various feature selection methods we first define some typical distances used to quantitatively measure the difference between two entities, such as data points and groups of data points. In the following, we assume there are N samples in the dataset, of which N_k samples belong to class C_k, i.e., $N_1 + \cdots + N_K = N$, and we define $P_k = N_k/N$ as the prior probability for any sample $\mathbf{x}$ randomly selected from the dataset to belong to C_k.

- **Distance between two data points**

  The distance between two points $\mathbf{x}$ and $\mathbf{y}$ in the d-dimensional feature space can be measured by the p-norm of the difference $\mathbf{x} - \mathbf{y}$ (Eq. (A.203) in Section A.4.1):

$$d_p(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_p = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p} \qquad 1 \leq p \leq \infty \tag{9.1}$$

  Specially, consider the following three cases:

  – $p = 1$, *city block or Manhattan distance*:

$$d_1(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_1 = \sum_{i=1}^{d} |x_i - y_i| \tag{9.2}$$

  – $p = 2$, *Euclidean distance*:

$$d_2(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \sqrt{\sum_{i=1}^{d} |x_i - y_i|^2} \tag{9.3}$$

  – $p = \infty$, the *Chebyshev distance*:

$$d_\infty(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_\infty = \max\{|x_1 - y_1|, \cdots, |x_d - y_d|\} \tag{9.4}$$

- **Intra-class distance**

  An intra-class distance measures how widely (or narrowly) all samples in each class C_k are distributed in the space, i.e., how tightly all data points in the class are clustered . It needs to be small for good separability.

– The *max diameter*:

$$d_{max}(C_k) = \max_{\mathbf{x},\mathbf{y}\in C_k} d(\mathbf{x},\mathbf{y}) \tag{9.5}$$

– The *average diameter*:

$$d_{average}(C_k) = \frac{1}{N_k(N_k-1)} \sum_{\mathbf{x},\mathbf{y}\in C_k, \mathbf{x}\neq\mathbf{y}} d(\mathbf{x},\mathbf{y}) \tag{9.6}$$

– The average distance from each sample to the mean vector of the cluster:

$$\frac{1}{N_k} \sum_{\mathbf{x}\in C_k} d(\mathbf{x},\mathbf{m}_k), \quad \text{where} \quad \mathbf{m}_k = \frac{1}{N_k} \sum_{\mathbf{x}\in C_k} \mathbf{x} \tag{9.7}$$

– The covariance of all samples in the class represents the tightness of the samples in the class:

$$\mathbf{\Sigma}_k = \frac{1}{N_k} \sum_{\mathbf{x}\in C_k} (\mathbf{x}-\mathbf{m}_k)(\mathbf{x}-\mathbf{m}_k)^T \tag{9.8}$$

and its determinant or trace can be used as such a scalar measurement of the tightness:

$$\det \mathbf{\Sigma}_k = \prod_{i=1}^{d} \lambda_i, \quad \text{tr}\, \mathbf{\Sigma}_k = \sum_{i=1}^{d} \lambda_i \tag{9.9}$$

As $\mathbf{\Sigma}_k$ is positive semi-definite, all of its eigenvalues λ_i are non-negative and so are its determinant and trace.

- **Distance between a point and a cluster/class of points**

The distance between a single point $\mathbf{x}$ and a cluster C_k can be defined differently depending on how the cluster is characterized.

– The *max and min-distances:* This distance assumes no additional knowledge of the cluster except its member data points.

$$d_{max}(\mathbf{x},C_k) = \max_{\mathbf{y}\in C_k} d(\mathbf{x},\mathbf{y}), \quad d_{min}(\mathbf{x},C_k) = \min_{\mathbf{y}\in C_k} d(\mathbf{x},\mathbf{y}) \tag{9.10}$$

– The *centroid distance:* This distance assumes each cluster C_k of points is characterized by their mean $\mathbf{m}_k$ representing their central location in the feature space.

$$d_{centroid}(\mathbf{x},C_k) = d_2(\mathbf{x},\mathbf{m}_k) = ||\mathbf{x}-\mathbf{m}_k||^2 = (\mathbf{x}-\mathbf{m}_k)^T(\mathbf{x}-\mathbf{m}_k) \tag{9.11}$$

– The *Mahalanobis distance:* This distance assumes each cluster C_k of points is characterized by their covariances in $\mathbf{\Sigma}_k$ representing how they are distributed (the tightness-looseness, the shape and orientation of their distribution) as well as their mean $\mathbf{m}_k$ for the central position.

$$d_M(\mathbf{x},C_k) = (\mathbf{x}-\mathbf{m}_k)^T \mathbf{\Sigma}_k^{-1} (\mathbf{x}-\mathbf{m}_k) \tag{9.12}$$

Note that this distance is positively related to the distance between

$\mathbf{x}$ and $\mathbf{m}_k$, but negatively related to $\mathbf{\Sigma}_k$, i.e., the more tightly they are clustered, the greater the distance $d_M(\mathbf{x}, C_k)$.

- **Inter-class distance**

  An intra-class distance measures the difference between two classes, it needs to be large for good separability.

  – The *max and min-distances*:

$$d_{max}(C_i, C_j) = \max_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} d(\mathbf{x}, \mathbf{y}), \qquad d_{min}(C_i, C_j) = \min_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} d(\mathbf{x}, \mathbf{y}) \tag{9.13}$$

  – The *average distance* between two clusters is the average of all pair-wise distances (e.g., Euclidean) between members of the two classes:

$$d_{average}(C_i, C_j) = \frac{1}{N_i N_j} \sum_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} d(\mathbf{x}, \mathbf{y}) \tag{9.14}$$

  – The *centroid distance* is the distance (e.g., Euclidean) between the centroids (mean vectors) of the two classes:

$$d_{centroid}(C_i, C_j) = d(\mathbf{m}_i, \mathbf{m}_j) \tag{9.15}$$

  – The *Bhattacharyya distance*:

$$d_B(C_i, C_j) = \frac{1}{4}(\mathbf{m}_i - \mathbf{m}_j)^T \left(\frac{\mathbf{\Sigma}_i + \mathbf{\Sigma}_j}{2}\right)^{-1} (\mathbf{m}_i - \mathbf{m}_j) + \log\left[\frac{|(\mathbf{\Sigma}_i + \mathbf{\Sigma}_j)/2|}{(|\mathbf{\Sigma}_i|\ |\mathbf{\Sigma}_j|)^{1/2}}\right] \tag{9.16}$$

  Similar to the the Mahalanobis distance, the first term is for the difference between the mean vectors of the two clusters but negatively related to the average of their covariances. The second term reflects the difference between the covariances of the two clusters, which is always positive due to the *AM-GM inequality* (algebraic mean vs geometric mean) of the arithmetic and geometric means:

$$\frac{1}{n}\sum_{i=1}^{n} x_i \geq \left(\prod_{i=1}^{n} x_i\right)^{1/n} \tag{9.17}$$

  Note that even when $\mathbf{m}_i = \mathbf{m}_j$ and therefore the first term is zero, the Bhattacharyya distance is still greater than zero if $\mathbf{\Sigma}_i \neq \mathbf{\Sigma}_j$.

The inter and intra-class distances can also defined based on a set of scatter matrices:

- Total scatter matrix:

  Thee total scatter matrix is the same as the covariance matrix of the entire dataset:

$$\mathbf{S}_T = \frac{1}{N}\sum_{\text{all } \mathbf{x}} (\mathbf{x} - \mathbf{m})(\mathbf{x} - \mathbf{m})^T = \frac{1}{N}\sum_{k=1}^{K}\sum_{\mathbf{x}\in C_k} (\mathbf{x} - \mathbf{m})(\mathbf{x} - \mathbf{m})^T \tag{9.18}$$

Due to the total mean $\sum_{\mathbf{X}} \mathbf{x} = \mathbf{m}$, $\text{rank}\,\mathbf{S}_T) \leq N-1$. The equality holds if all N samples are independent.

- Within-class (intra-class) scatter matrix:

$$\mathbf{S}_W = \sum_{k=1}^{K} P_k \boldsymbol{\Sigma}_k = \sum_{k=1}^{K} \frac{N_k}{N} \boldsymbol{\Sigma}_k = \frac{1}{N} \sum_{k=1}^{K} \sum_{\mathbf{x} \in C_k} (\mathbf{x} - \mathbf{m}_k)(\mathbf{x} - \mathbf{m}_k)^T \quad (9.19)$$

Due to the K means $\mathbf{m}_k = \sum_{\mathbf{x} \in C_k} \mathbf{x}/N_k, \quad (k = 1, \cdots, K)$ as K constraints, $\text{rank}(\mathbf{S}_W) = \sum_{k=1}^{K} (N_k - 1) \leq N - K$. Again, the equality holds if all N samples are independent.

- Between-class (inter-class) scatter matrix:

$$\mathbf{S}_B = \sum_{k=1}^{K} P_k (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T = \sum_{k=1}^{K} \frac{N_k}{N} (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T \quad (9.20)$$

Due to total mean $\mathbf{m} = \sum_{k=1}^{K} N_k \mathbf{m}_k / N$ as an additional constraint, $\text{rank}(\mathbf{S}_B) \leq K - 1$.

We can show that the total scatteredness of the dataset is contributed by the within-class and between-class scatteredness, i.e., $\mathbf{S}_T = \mathbf{S}_W + \mathbf{S}_B$:

$$\begin{aligned}
\mathbf{S}_T &= \frac{1}{N} \sum_{k=1}^{K} \sum_{\mathbf{x} \in C_k} (\mathbf{x} - \mathbf{m})(\mathbf{x} - \mathbf{m})^T = \frac{1}{N} \sum_{k=1}^{K} \sum_{\mathbf{x} \in C_k} (\mathbf{x} - \mathbf{m}_k + \mathbf{m}_k - \mathbf{m})(\mathbf{x} - \mathbf{m}_k + \mathbf{m}_k - \mathbf{m})^T \\
&= \frac{1}{N} \sum_{k=1}^{K} \sum_{\mathbf{x} \in C_k} [(\mathbf{x} - \mathbf{m}_k)(\mathbf{x} - \mathbf{m}_k)^T + (\mathbf{x} - \mathbf{m}_k)(\mathbf{m}_k - \mathbf{m})^T \\
&\quad + (\mathbf{m}_k - \mathbf{m})(\mathbf{x} - \mathbf{m}_k)^T + (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T] \\
&= \frac{1}{N} \sum_{k=1}^{K} \sum_{\mathbf{x} \in C_k} (\mathbf{x} - \mathbf{m}_k)(\mathbf{x} - \mathbf{m}_k)^T + \frac{1}{K} \sum_{k=1}^{K} \sum_{\mathbf{x} \in C_k} (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T \\
&= \sum_{k=1}^{K} \frac{N_k}{N} \boldsymbol{\Sigma}_k + \sum_{k=1}^{K} \frac{N_k}{N} (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T = \mathbf{S}_W + \mathbf{S}_B
\end{aligned} \quad (9.21)$$

where the two middle terms are both zero as $\sum_{\mathbf{x} \in C_k} (\mathbf{x} - \mathbf{m}_k) = \mathbf{0}$.

For better separability, we want the within-class scatteredness to be small but the between-class scatteredness to be large. However, as these scatter matrices cannot be directly compared in terms of their sizes, we instead consider their traces (or determinants) as the scalar measurements for the separability:

$$\begin{aligned}
J_B &= \text{tr}\ \mathbf{S}_B = \text{tr}\ \left[\sum_{k=1}^{C} P_k (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T \right] \\
&= \sum_{i=1}^{C} P_k\ \text{tr}\left[(\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T \right] = \sum_{i=1}^{C} P_k\ ||\mathbf{m}_k - \mathbf{m}||^2
\end{aligned} \quad (9.22)$$

and

$$J_W = \text{tr } \mathbf{S}_W = \text{tr}\left[\sum_{k=1}^{K} P_k \sum_{\mathbf{x}\in C_k} (\mathbf{x}-\mathbf{m}_k)(\mathbf{x}-\mathbf{m}_k)^T\right]$$
$$= \sum_{k=1}^{K} P_k \sum_{\mathbf{x}\in C_k} \text{tr}\left[(\mathbf{x}-\mathbf{m}_k)(\mathbf{x}-\mathbf{m}_k)^T\right] = \sum_{k=1}^{K} P_k \sum_{\mathbf{x}\in C_k} ||\mathbf{x}-\mathbf{m}_k||^2 \quad (9.23)$$

We see that J_B is the weighted average of the Euclidean distances between $\mathbf{m}_k$ and total $\mathbf{m}$ for the entire dataset containing all K classes, and J_W is the weighted average of the Euclidean distances between all $\mathbf{x} \in C_k$ to $\mathbf{m}_k$ for all K classes $C_1, \cdots, C_K$. We also recall that the trace or determinant of a matrix is the sum or product of all of its eigenvalues (Eq. (A.105) in Section A.3.1).

To achieve a high separability, we desire to maximize $\text{tr}\,\mathbf{S}_B$ and minimize $\text{tr}\,\mathbf{S}_W$. In the same space with a constant $\mathbf{S}_T = \mathbf{S}_B + \mathbf{S}_W$, maximizing $\text{tr}\,\mathbf{S}_B$ is equivalent to minimizing $\text{tr}\,\mathbf{S}_W$. However, to measure the separability across different spaces with different $\mathbf{S}_T$, we need to normalize $\mathbf{S}_B$ by either $\mathbf{S}_T$ or equivalently $\mathbf{S}_W$, and redefine the objective function as either of the following:

$$J_{B/W} = \text{tr}(\mathbf{S}_W^{-1}\mathbf{S}_B) = \text{tr}(\mathbf{S}_B\mathbf{S}_W^{-1}), \qquad J_{B/T} = \text{tr}(\mathbf{S}_T^{-1}\mathbf{S}_B) = \text{tr}(\mathbf{S}_B\mathbf{S}_T^{-1}) \quad (9.24)$$

To reduce the dimensionality of the feature space while maximally conserving the separability in the original d-D space, we can select $d' < d$ of the d original features that maximize either $J_{B/W}$ of $J_{B/T}$ as an objective function.

Problems

In many of the problems in this homework and subsequent ones it is required to develop code to carry out feature extraction, dimensionality reduction, and classification/clustering, to be tested on both real and synthetic datasets. The sample code below can be used to generate a synthetic dataset containing multiple classes each composed of a set of normally distributed data points, based on user specified parameters including dimensionality d, number of classes K, and number of samples in each class (adding up to the total number of samples N), as well as the means and covariances (for the positions and shape of the normal distribution respectively). The code generates a $d \times N$ array $\mathbf{X}$ for all data points and an N-D vector $\mathbf{y}$ for the labelings, to be used as the ground truth for any supervised learning algorithm. Note that the repeatability of the algorithm is desired, a fixed seed for the random number generator needs to be specified.

```
function [X y]=sampleDataset
    K=4;                % number of classes
    d=3;                % dimensionality of dataset
    rng(1);             % seed for random number generator
    Nk=[150 200 250];   % number of samples in each class
```

```
    N=sum(Nk);           % totall number of samples
    mu=zeros(d,K);       % mean vectors for all classes
    Sigma=zeros(d,d,K);  % covariance matrices for all classes
    Sigma(:,:,1)=eye(3); % covariance matrix for each class
    Sigma(:,:,2)=eye(3); % feel free to specify any other form
    Sigma(:,:,3)=eye(3);
    mu(:,1)=[0; 0; 0];   % specify mean vector for each class
    mu(:,2)=[0; 0; 1];
    mu(:,3)=[0; 1; 0];
    mu=mu*4;             % scaling factor (for varying separability)
    X=[];                % for all data samples (d by N matrix)
    y=[];                % for their labelings (1 by N vector)
    for k=1:K            % for each of K classes
        Xk=mvnrnd(mu(:,k),Sigma(:,:,k), Nk(k))';
                         % generate normally distributed dataset
        X=[X Xk];        % concatenate all samples to X
        y=[y k*ones(1,Nk(k))];  % concatenate their labeling to y
        switch k
            case 1
                color='r';
            case 2
                color='g';
            case 3
                color='b';
        end
        scatter3(Xk(1,:),Xk(2,:),Xk(3,:),[],color,'.') % visualize
        hold on
    end
    hold off
    data=[X; y];
    save('data34.txt','data','-ASCII') % save dataset as ASCII file
end
```

The code below can be used to read in the data file generated by the code above, and any dataset composed of data array $\mathbf{X}$ and the labeling vector $\mathbf{y}$ with data points in arbitrary order.

```
    data=load('data34.txt','data');  % read disk file into data
    d=size(data,1)-1       % dimensionality of data
    N=size(data,2);        % total number of samples
    X=data(1:d,:);         % all N d-dimensional sample vectors
    y=data(d+1,:);         % and their labelings
    K=length(unique(y));   % number of classes
    for k=1:K              % for each of the K classes
        idx=find(y==k);    % indices of samples in class k
```

```
    Xk=X(:,idx)        % collect all samples in class k
    mean(Xk')          % mean vector of class k
    cov(Xk')           % covariance matrix of class k
end
```

In addition to synthetic data, some real datasets are also used in both homework problems and examples in the current and future chapters, such as the iris flower dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set), which was first introduced in 1936 by British statistician and biologist Ronald Fisher, and has been widely used for benchmarking various classification algorithms in recent decades. This dataset contains 150 samples of iris flowers of three different species (setosa, virginica and versicolor) each containing 50 samples characterized by four features, the length and width of sepals and petals of the flower. This dataset can be read into your Metlab code by a built-in Metlab function:

```
X=iris_dataset;        % read in iris data
[d, N]=size(data);     % dimensionality and number of samples
y=[ones(1,50) 2*ones(1,50) 3*ones(1,50)];  % data labelings
```

1. Generate and visualize multi-class synthetic data generated by the code above. Try different parameter values (e.g., $d = 2, 3$, and $K = (3, 4, 5, \cdots)$), and experiment different mean vectors and covariance matrices to see how the locations and shapes of distributions of the data points are effected.
2. Generate a dataset (e.g., $d = 3$ and $K = 5$) and find all $K(K-1)/2$ pair-wise distances $\{d_p(\mathbf{m}_i, \mathbf{m}_j),\ i = 2, \cdots, K,\ j = 1, \cdots, i-1\}$ between the mean vectors of any two of different classes for each value of $p = 1,\ 2,\ \infty$. Display your results as the lower triangular matrix (all components below the main diagonal of a $K \times K$ matrix).
3. Find all pair-wise Bhattacharyya distances between any two classes, and show them as the lower triangular matrix, and compare them with those obtained in the previous problem.
4. Find the between-class, within-class and total scatter matrices and verify they do satisfy $\mathbf{S}_B + \mathbf{S}_W = \mathbf{S}_T$.
5. Repeat all problems above for the iris dataset.

10 Principal Component Analysis

The *Principal component analysis (PCA)* is a method widely used for visualizing, processing and analyzing large datasets of high dimensionalities. The PCA is capable of significantly reducing the dimensionality of the dataset, while still maximally keeping the signal variation or information of interest in the dataset measured in certain way, so that the subsequent processing and analysis of the dataset can be carried out much more effectively and efficiently.

As always, we assume the given dataset is in the form of a $d \times N$ matrix $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, of which each column vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ is a data sample, a point in the d-dimensional feature space. In the context of pattern classification, each component x_i of $\mathbf{x}$ is some measurement of one of the d features of a pattern. Given $\mathbf{X}$, the PCA carries out a special orthogonal transformation $\mathbf{y} = \mathbf{A}^T\mathbf{x}$, called *Karhunen–Loève transform (KLT)*, in such a way that the components in the resulting vector $\mathbf{y}$ are completely decorrelated and the energy or information contained in the data is redistributed optimally so that most of the energy is concentrated in a small number of components called *principal components*. Based on these two highly desirable properties, the KLT can be used for data compression in data processing and feature extraction in pattern recognition.

10.1 Covariance and Correlation

In PCA, each pattern $\mathbf{x} = [x_1, \cdots, x_d]^T$ is treated as a random vector of which each component x_i is a random variable with mean and variance

$$\begin{aligned} \mu_i &= \mathrm{E}[\mathbf{x}_i] = \int x_i \, p(x_i) \, dx_i \\ \sigma_i^2 &= \mathrm{E}[(x_i - \mu_i)^2] = \mathrm{E}[x_i^2] - \mu_i^2 = \int x_i^2 \, p(x_i) \, dx_i - \mu_{x_i}^2 \\ &\qquad (i = 1, \cdots, d) \end{aligned} \tag{10.1}$$

and the *covariance* of x_i and x_j is:

$$\begin{aligned} \sigma_{ij}^2 &= \mathrm{E}[(x_i - \mu_i)(x_j - \mu_j)] = \mathrm{E}[x_i x_j] - \mu_i \mu_j \\ &= \int\int x_i x_j \, p(x_i, x_j) \, dx_i \, dx_j - \mu_i \mu_j \qquad (i, j = 1, \cdots, d) \end{aligned} \tag{10.2}$$

The mean vector and covariance matrix of $\mathbf{x}$ are:

$$\begin{aligned}\mathbf{m}_x &= \mathrm{E}[\mathbf{x}] = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_d \end{bmatrix} \\ \boldsymbol{\Sigma}_x &= \mathrm{E}[(\mathbf{x}-\mathbf{m}_x)(\mathbf{x}-\mathbf{m}_x)^T] = \mathrm{E}[\mathbf{x}\mathbf{x}^T] - \mathbf{m}_x\mathbf{m}_x^T \\ &= \begin{bmatrix} \sigma_{11}^2 & \cdots & \sigma_{1d}^2 \\ \vdots & \ddots & \vdots \\ \sigma_{d1}^2 & \cdots & \sigma_{dd}^2 \end{bmatrix}\end{aligned} \tag{10.3}$$

Usually the joint probability density function $p(\mathbf{x}) = p(x_1, \cdots, x_d)$ of the random vector $\mathbf{x}$ is unknown, and the mean vector $\mathbf{m}_x$ and covariance matrix $\boldsymbol{\Sigma}_x$ of $\mathbf{x}$ can only be estimated by the method of *maximum likelihood estimation (MLE)* (Section B.3.3) based on a set of observed data samples in data set $\mathbf{X}$:

$$\hat{\mathbf{m}}_x = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n, \qquad \hat{\boldsymbol{\Sigma}} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \hat{\mathbf{m}})(\mathbf{x}_n - \hat{\mathbf{m}})^T = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^T - \hat{\mathbf{m}}\hat{\mathbf{m}}^T \tag{10.4}$$

Note that the rank of the $d \times d$ estimated covariance matrix $\hat{\boldsymbol{\Sigma}}$ is at most $N-1$, due to the N samples in the dataset $\mathbf{X}$, assumed to be are independent, and the additional constraint:

$$\sum_{n=1}^{N}(\mathbf{x}_n - \hat{\mathbf{m}}) = \sum_{n=1}^{N}\mathbf{x}_n - N\hat{\mathbf{m}} = \mathbf{0} \tag{10.5}$$

The variance $\sigma_i^2 = \mathrm{E}[(x_i - \mu_{x_i})^2]$ can be treated as the dynamic energy contained in x_i, or the amount of information carried in x_i, while the trace $\mathrm{tr}\,\boldsymbol{\Sigma}_x = \sum_{i=1}^{d}\sigma_i^2$ can be considered as the total amount of dynamic energy contained in $\mathbf{x}$. Also, the covariance $\sigma_{ij}^2 = \mathrm{E}[(x_i - \mu_{x_i})(x_j - \mu_{x_j})]$ can be considered as the *mutual energy*, a measure of the correlation between x_i and x_j. By normalizing the covariance σ_{ij}^2, we get the *correlation coefficient* between x_i and x_j:

$$\rho_{ij} = \frac{\sigma_{ij}^2}{\sqrt{\sigma_i^2\,\sigma_j^2}} = \frac{\sigma_{ij}^2}{\sigma_i\,\sigma_j} \tag{10.6}$$

that measures how the two random variables x_i and x_j are correlated.

- $\rho_{ij} = \pm 1$ or $|\rho_{ij}| = 1$: x_i and x_j are maximally correlated. The information contained in the two variables is completely redundant, i.e., given the value of one of them, the value of the other is known.
- $-1 < \rho_{ij} < 1$ or $|\rho_{ij}| < 1$: x_i and x_j are correlated to different extents. The information they each carry has certain redundancy.
- $\rho_{ij} = 0$: x_i and x_j are uncorrelated. They each carry their own independent information with no redundancy.

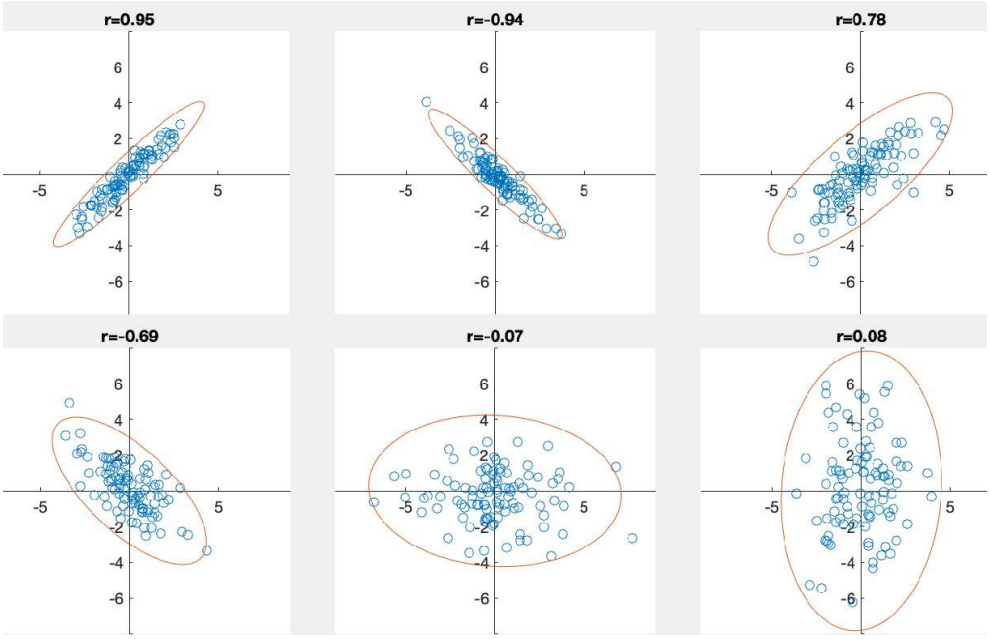


Figure 10.1 Normally distributed datasets with different covariance matrices

We see that the correlation coefficient ρ_{ij} that measures how much two random variables are correlated also measures the redundancy of the information they carry. When there exists some data redundancy in the data, it is possible to carry out data compression by some method such as the PCA based on the covariance $\mathbf{\Sigma}$ to reduce the data redundancy, so that the data size can be significantly reduced while the information (dynamic energy) contained in the data is still mostly conserved.

For example, six normally distributed 2-D datasets are generated with zero mean and the following covariance matrices:

$$\mathbf{\Sigma}_1 = \begin{bmatrix} 1.0 & 0.95 \\ 0.95 & 1.0 \end{bmatrix}, \quad \mathbf{\Sigma}_2 = \begin{bmatrix} 1.0 & -0.95 \\ -0.95 & 1.0 \end{bmatrix}, \quad \mathbf{\Sigma}_2 = \begin{bmatrix} 1.0 & 0.7 \\ 0.7 & 1.0 \end{bmatrix}$$

$$\mathbf{\Sigma}_4 = \begin{bmatrix} 1.0 & -0.7 \\ -0.7 & 1.0 \end{bmatrix}, \quad \mathbf{\Sigma}_5 = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{\Sigma}_6 = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}$$

Fig. 10.1 shows the 2-D data points together with the contour line (isoline) of the corresponding normal distribution based on each of the six covariance matrices. The correlation coefficient is also shown on top of each dataset.

10.2 Karhunen-Loève Transformation

A data point in a d-dimensional space is represented by a vector $\mathbf{x} = [x_1, \cdots, x_d]^T$, of which the components are the coordinates along the d standard orthonormal basis vectors $\{\mathbf{e}_1, \cdots, \mathbf{e}_d\}$ that span the d-D space:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} + \cdots + x_d \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} = \sum_{i=1}^{d} x_i \mathbf{e}_i$$
$$= \begin{bmatrix} \mathbf{e}_1 & \cdots & \mathbf{e}_d \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} = \mathbf{I}\mathbf{x} \tag{10.7}$$

where $\mathbf{I} = [\mathbf{e}_1, \cdots, \mathbf{e}_d]$ is the identity matrix, and the ith component $x_i = \mathbf{e}_i^T \mathbf{x}$ is the projection of $\mathbf{x}$ onto the ith standard basis vector $\mathbf{e}_i$.

However, as shown in Fig. 10.2, the space can also be spanned by aother orthonormal basis as the column vectors in any orthogonal matrix $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_d]$ satisfying

$$\mathbf{a}_i^T \mathbf{a}_j = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{10.8}$$

and the vector can also be equivalently represented as a weighted sum of these basis vectors:

$$\mathbf{x} = \sum_{i=1}^{d} x_i \mathbf{e}_i = \sum_{i=1}^{d} y_i \mathbf{a}_i = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_d \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_d \end{bmatrix} = \mathbf{A}\mathbf{y} \tag{10.9}$$

where $\mathbf{y} = [y_1, \cdots, y_d]^T$ is a vector containing all d coordinates in the directions of the basis vector $\{\mathbf{a}_i, \cdots, \mathbf{a}_d\}$.

The orthogonal matrix $\mathbf{A}$ satisfies $\mathbf{A}^T = \mathbf{A}^{-1}$, i.e., $\mathbf{A}\mathbf{A}^T = \mathbf{A}^T\mathbf{A} = \mathbf{I}$. Premultiplying $\mathbf{A}^T$ on both sides of the equation above, we get $\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{A}\mathbf{y} = \mathbf{A}^{-1}\mathbf{A}\mathbf{y} = \mathbf{y}$, i.e.,

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_d \end{bmatrix} = \mathbf{A}^T\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_d^T \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{x} \\ \vdots \\ \mathbf{a}_d^T\mathbf{x} \end{bmatrix} \tag{10.10}$$

where the ith coordinate $y_i = \mathbf{a}_i^T\mathbf{x}$ is the projection of $\mathbf{x}$ onto the ith basis vector $\mathbf{a}_i$. The two equations above form an orthogonal transform pair:

$$\begin{cases} \mathbf{y} = \mathbf{A}^T\mathbf{x} & \text{forward transform} \\ \mathbf{x} = \mathbf{A}\mathbf{y} & \text{inverse transform} \end{cases} \tag{10.11}$$

The basis vectors in $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_d]$ can be considered as a rotated version

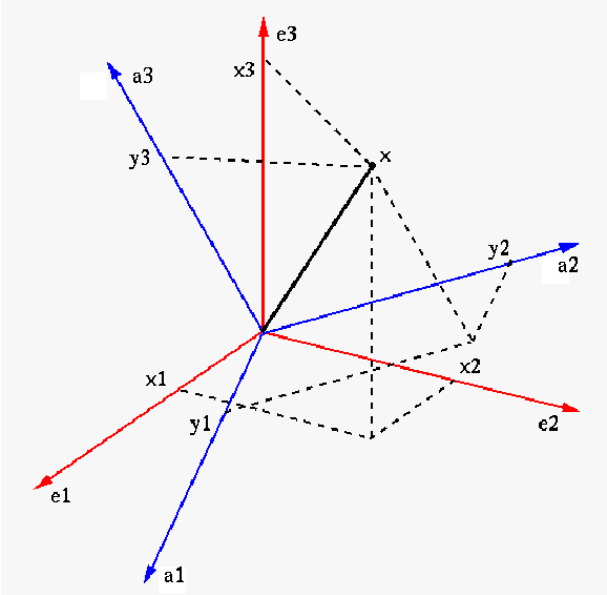


Figure 10.2 Orthogonal transform as a rotation of the coordinate system

of the standard basis in $\mathbf{I} = [\mathbf{e}_1, \cdots, \mathbf{e}_d]$, and the norm or length of the vector before and after the transform remain the same:

$$||\mathbf{x}||^2 = \mathbf{x}^T\mathbf{x} = \left(\sum_{i=1}^{d} y_i\mathbf{a}_i\right)^T \left(\sum_{j=1}^{d} y_j\mathbf{a}_j\right) = \sum_{i=1}^{d}\sum_{j=1}^{d} y_i y_j\, \mathbf{a}_i^T\mathbf{a}_j = \sum_{i=1}^{d} y_i^2 = ||\mathbf{y}||^2 \tag{10.12}$$

If $\mathbf{x}$ is treated as a random vector, then its linear transform $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ is also a random vector, with the following mean and covariance:

$$\begin{aligned} \mathbf{m}_y &= \mathrm{E}[\mathbf{y}] = \mathrm{E}[\mathbf{A}^T\mathbf{x}] = \mathbf{A}^T\,\mathrm{E}[\mathbf{x}] = \mathbf{A}^T\mathbf{m}_x \\ \mathbf{\Sigma}_y &= \mathrm{E}[\mathbf{y}\mathbf{y}^T] - \mathbf{m}_y\mathbf{m}_y^T = \mathrm{E}[\mathbf{A}^T\mathbf{x}\mathbf{x}^T\mathbf{A}] - \mathbf{A}^T\mathbf{m}_x\mathbf{m}_x^T\mathbf{A} \\ &= \mathbf{A}^T[\mathrm{E}[\mathbf{x}\mathbf{x}^T] - \mathbf{m}_x\mathbf{m}^T]\mathbf{A} = \mathbf{A}^T\mathbf{\Sigma}_x\mathbf{A} \end{aligned} \tag{10.13}$$

The *Karhunen-Loève Transform (KLT)* is just a special case of the orthogonal transforms considered above, in the same form of $\mathbf{y} = \mathbf{V}^T\mathbf{x}$, where the orthogonal transform matrix $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_d]$ is the eigenvector matrix of the covariance matrix $\mathbf{\Sigma}_x$ of $\mathbf{x}$, composed of the d normalized eigenvectors of $\mathbf{\Sigma}_x$. As in general the covariance matrix $\mathbf{\Sigma}_x$ is symmetric and positive definite, its eigenvalues $\{\lambda_1, \cdots, \lambda_d\}$ are real and positive, and its eigenvectors are orthogonal

$$\lambda_d \geq \cdots \geq \lambda_1 \geq 0, \qquad \mathbf{v}_i^T\mathbf{v}_j = \delta_{ij} = \begin{cases} 0 & i \geq j \\ 1 & i = j \end{cases} \qquad (i, j = 1, \cdots, d) \tag{10.14}$$

i.e., $\mathbf{V}$ is an orthogonal matrix satisfying $\mathbf{V}^T = \mathbf{V}^{-1}$ or $\mathbf{V}^T\mathbf{V} = \mathbf{I}$.

The d eigenvalues $\{\lambda_1, \cdots, \lambda_d\}$ and the corresponding eigenvectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_d\}$ can then be found by solving the eigenequations:

$$\boldsymbol{\Sigma}_x \mathbf{v}_i = \lambda_i \mathbf{v}_i \qquad (i = 1, \cdots, d) \tag{10.15}$$

Note that the eigenvector $\mathbf{v}_i$ is not unique, as the eigenequation still holds when $\mathbf{v}_i$ is arbitrarily scaled by any constant. The d eigenequations above can be combined to be expressed in matrix form:

$$\boldsymbol{\Sigma}_x \mathbf{V} = \boldsymbol{\Sigma}_x [\mathbf{v}_1, \cdots, \mathbf{v}_d] = [\mathbf{v}_1, \cdots, \mathbf{v}_d] \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_d \end{bmatrix} = \mathbf{V}\boldsymbol{\Lambda} \tag{10.16}$$

where $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_d)$ is a diagonal matrix.

Premultiplying $\mathbf{v}_i^T$ on both sides of the eigenequation in Eq. (10.15), we get:

$$\mathbf{v}_i^T \boldsymbol{\Sigma}_x \mathbf{v}_i = \lambda_i \mathbf{v}_i^T \mathbf{v}_i = \lambda_i \qquad (i = 1, \cdots, d) \tag{10.17}$$

or in matrix form, premultiplying $\mathbf{V}^T = \mathbf{V}^{-1}$ on both sides of Eq. (10.16), we get

$$\mathbf{V}^T \boldsymbol{\Sigma}_x \mathbf{V} = \mathbf{V}^{-1}\mathbf{V}\boldsymbol{\Lambda} = \boldsymbol{\Lambda} \tag{10.18}$$

Taking inverse on both sides, we also get

$$(\mathbf{V}^T \boldsymbol{\Sigma}_x \mathbf{V})^{-1} = \mathbf{V}^{-1}\boldsymbol{\Sigma}_x^{-1}(\mathbf{V}^T)^{-1} = \mathbf{V}^T \boldsymbol{\Sigma}_x^{-1} \mathbf{V} = \boldsymbol{\Lambda}^{-1} \tag{10.19}$$

We see that $\boldsymbol{\Sigma}_x^{-1}$ and $\boldsymbol{\Sigma}_x$ share the same eigenvectors, their eigenvalues are reciprocal of each other, and they are both diagonalized by the orthogonal eigenvector matrix $\mathbf{V}$ to become $\boldsymbol{\Lambda}$ and $\boldsymbol{\Lambda}^{-1}$ respectively.

Based on the orthogonal eigenvector matrix $\mathbf{V}$, the KLT is defined as:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_d \end{bmatrix} = \mathbf{V}^T \mathbf{x} = \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_d^T \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{v}_1^T \mathbf{x} \\ \vdots \\ \mathbf{v}_d^T \mathbf{x} \end{bmatrix} \tag{10.20}$$

of which the ith component $y_i = \mathbf{v}_i^T \mathbf{x}$ is the projection of vector $\mathbf{x}$ onto the ith eigenvector $\mathbf{v}_i$. Premultiplying $\mathbf{V} = (\mathbf{V}^T)^{-1}$ on both sides, we get

$$\mathbf{V}\mathbf{y} = \mathbf{V}\mathbf{V}^T\mathbf{x} = \mathbf{V}\mathbf{V}^{-1}\mathbf{x} = \mathbf{x} \tag{10.21}$$

The two equations above form the KLT transform pair:

$$\begin{cases} \mathbf{y} = \mathbf{V}^T \mathbf{x} & \text{forward transform} \\ \mathbf{x} = \mathbf{V}\mathbf{y} & \text{inverse transform} \end{cases} \tag{10.22}$$

10.3 Optimality of KLT

The KLT is the optimal orthogonal transform among all possible orthogonal transforms $\mathbf{y} = \mathbf{A}^T \mathbf{x}$ based on any orthogonal matrix $\mathbf{A}$ satisfying $\mathbf{A}^T\mathbf{A} = \mathbf{I}$,

in the sense that the KLT completely decorrelates the signal, and it maximally compacts the energy (information) contained in the signal, as shown below.

- **KLT Completely Decorrelates Signal Components**

  The mean vector and covariance matrix of the random vector $\mathbf{y} = \mathbf{V}^T\mathbf{x}$ after the KLT transform can found as

$$\mathbf{m}_y = \mathrm{E}[\mathbf{y}] = \mathrm{E}[\mathbf{V}^T\mathbf{x}] = \mathbf{V}^T\,\mathrm{E}[\mathbf{x}] = \mathbf{V}^T\mathbf{m}_x \tag{10.23}$$

$$\begin{aligned}\mathbf{\Sigma}_y &= \mathrm{E}[(\mathbf{y}-\mathbf{m}_y)(\mathbf{y}-\mathbf{m}_y)^T] = \mathrm{E}[\mathbf{V}^T(\mathbf{x}-\mathbf{m}_x)][\mathbf{V}^T(\mathbf{x}-\mathbf{m}_x)]^T \\ &= \mathbf{V}^T E\left[(\mathbf{x}-\mathbf{m}_x)(\mathbf{x}-\mathbf{m}_x)^T\right]\mathbf{V} = \mathbf{V}^T\mathbf{\Sigma}_x\mathbf{V} = \mathbf{\Lambda} \\ &= \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_d \end{bmatrix} = \begin{bmatrix} \sigma_{y_1}^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_{y_d}^2 \end{bmatrix}\end{aligned} \tag{10.24}$$

  Based on the fact that the covariance matrix $\mathbf{\Sigma}_y$ is a diagonal matrix, we have the following two observations:

  – The zero off-diagonal component $\sigma_{ij}^2 = 0$ $(i \neq j)$ for the covariance between any two components y_i and y_j indicates they are completely decorrelated after the KLT, i.e., each component carries its own independent information;

  – The nonzero diagonal component for the variance $\lambda_i = \sigma_i^2 = \mathrm{E}[(y_i - \mu_{y_i})^2]$ $(i = 1, \cdots, c)$ of the ith component y_i represents the dynamic energy or amount of information contained in y_i. As $\mathbf{\Sigma}_x$ and $\mathbf{\Sigma}_y = \mathbf{\Lambda}$ share the same eigenvalues $\{\lambda_1, \cdots, \lambda_d\}$, and therefore have the same trace

$$\mathcal{E}_x = \mathrm{tr}\,\mathbf{\Sigma}_x = \sum_{i=1}^{d} \lambda_i = \mathrm{tr}\,\mathbf{\Sigma}_y = \mathcal{E}_y \tag{10.25}$$

    We see that the total amount of energy or information contained in the signal $\mathbf{x}$ before the KLT and $\mathbf{y}$ after the KLT remains the same, i.e., the energy or information contained in the signal is conserved by the KLT. This is generally true for all orthogonal transforms $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ (such as the Fourier transform), due to the fact that the eigenvalues remain the same under any orthogonal transform $\mathbf{B} = \mathbf{R}^T\mathbf{A}\mathbf{R}$, i.e., matrices $\mathbf{A}$ and $\mathbf{B}$ share the same eigenvalues, and thereby the same trace and determinant.

- **KLT Optimally Compacts Signal Energy**

  While the total energy contained in the signal is conserved by any orthogonal transform, the distribution of this energy among the d components before and after the transform may be very different. We can show that the KLT is optimal in the sense that it redistributes the total energy in such a way that it is maximally compacted into a subset of components of $\mathbf{y} = \mathbf{V}^T\mathbf{x}$.

Consider a generic orthogonal transform $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ based on an arbitrary orthogonal matrix $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_d]$ satisfying $\mathbf{A}^T = \mathbf{A}^{-1}$:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_d \end{bmatrix} = \mathbf{A}^T\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_d^T \end{bmatrix} \mathbf{x} \tag{10.26}$$

We also have

$$y_i = \mathbf{a}_i^T\,\mathbf{x}, \qquad \mu_{y_i} = E(y_i) = \mathbf{a}_i^T\,E(\mathbf{x}) = \mathbf{a}_i^T\mathbf{m}_x, \qquad (i = 1, \cdots, d) \tag{10.27}$$

We then consider the energy contained in $d' < d$ out of the d components of $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ expressed as a function of the transform matrix $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_d]$:

$$\begin{aligned} \mathcal{E}'(\mathbf{A}) &= \sum_{i=1}^{d'} \sigma_{y_i}^2 = \sum_{i=1}^{d'} E\left[(y_i - \mu_{y_i})^2\right] = \sum_{i=1}^{d'} E\left[\mathbf{a}_i^T(\mathbf{x} - \mathbf{m}_{x_i})\,(\mathbf{x} - \mathbf{m}_{x_i})^T\mathbf{a}_i\right] \\ &= \sum_{i=1}^{d'} \mathbf{a}_i^T E\left[(\mathbf{x} - \mathbf{m}_{x_i})(\mathbf{x} - \mathbf{m}_{x_i})^T\right]\mathbf{a}_i^T = \sum_{i=1}^{d'} \mathbf{a}_i^T\mathbf{\Sigma}_x\mathbf{a}_i \end{aligned} \tag{10.28}$$

We can find the optimal orthogonal transform matrix $\mathbf{A}$ that maximizes this partial energy $\mathcal{E}'$ by solving the following optimization problem with the constraint that the column vectors of $\mathbf{A}$ are normalized with $\mathbf{a}_j^T\mathbf{a}_j = 1$:

$$\begin{cases} \text{maximize: } \mathcal{E}'(\mathbf{A}) = \mathcal{E}'(\mathbf{a}_1, \cdots, \mathbf{a}_{d'}) \\ \text{subject to: } \mathbf{a}_j^T\mathbf{a}_j = 1 \quad (j = 1, \cdots, d') \end{cases} \tag{10.29}$$

The Lagrangian function of this constrained optimization problem is

$$\mathcal{L}(\mathbf{A}) = \mathcal{E}'(\mathbf{A}) - \sum_{j=1}^{d'} \lambda_j(\mathbf{a}_j^T\mathbf{a}_j - 1) \tag{10.30}$$

To find the optimal $\mathbf{A}$, we set partial derivative with respect to each $\mathbf{a}_i$ $(i = 1, \cdots, d')$ to zero:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{a}_i}\mathcal{L}(\mathbf{A}) &= \frac{\partial}{\partial \mathbf{a}_i}\left[\sum_{j=1}^{d'}(\mathbf{a}_j^T\mathbf{\Sigma}_x\mathbf{a}_j - \lambda_j\mathbf{a}_j^T\mathbf{a}_j + \lambda_j)\right] \\ &= \frac{\partial}{\partial \mathbf{a}_i}\left(\mathbf{a}_i^T\mathbf{\Sigma}_x\mathbf{a}_i - \lambda_i\mathbf{a}_i^T\mathbf{a}_i\right) = 2\mathbf{\Sigma}_x\mathbf{a}_i - 2\lambda_i\mathbf{a}_i = \mathbf{0} \end{aligned} \tag{10.31}$$

The resulting equation $\mathbf{\Sigma}_x\mathbf{a}_i = \lambda_i\mathbf{a}_i$ happens to be the eigenequation of the covariance matrix $\mathbf{\Sigma}_x$, and the column vectors of $\mathbf{A}$ must be the eigenvectors of $\mathbf{\Sigma}_x$, i.e., the optimal transform is the KLT $\mathbf{A} = \mathbf{V}$.

$$\mathcal{E}'(\mathbf{V}) = \sum_{i=1}^{d'} \mathbf{v}_i^T\mathbf{\Sigma}_x\mathbf{v}_i = \sum_{i=1}^{d'} \lambda_i \le \sum_{i=1}^{d} \lambda_i = \mathcal{E}_y = \mathcal{E}_x \tag{10.32}$$

To maximize the partial energy $\mathcal{E}'(\mathbf{V})$ contained in the d' components we carry out the KLT based on the eigenvectors $\mathbf{v}_i$ corresponding to the d' largest eigenvalues of $\mathbf{\Sigma}_x$: $\lambda_1 \geq \lambda_2 \geq \cdots \lambda_{d'} \cdots \geq \lambda_d$.

10.4 Geometric Interpretation of the KLT

The optimality of the KLT discussed above can be demonstrated geometrically, based on the assumption that the random vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ has a normal probability density function:

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{m}_x, \mathbf{\Sigma}_x) = \frac{1}{(2\pi)^{d/2}\,|\mathbf{\Sigma}_x|^{1/2}}\ \exp\left[-\frac{1}{2}(\mathbf{x}-\mathbf{m}_x)^T\mathbf{\Sigma}_x^{-1}(\mathbf{x}-\mathbf{m}_x)\right] \tag{10.33}$$

with mean vector $\mathbf{m}_x$ and covariance matrix $\mathbf{\Sigma}_x$. The shape of this normal distribution in the d-dimensional space can be represented by an iso-hypersurface in the space determined by

$$\mathcal{N}(\mathbf{m}_x, \mathbf{\Sigma}_x) = c_0 \tag{10.34}$$

where c_0 is some constant, or equivalently

$$(\mathbf{x}-\mathbf{m}_x)^T\mathbf{\Sigma}_x^{-1}(\mathbf{x}-\mathbf{m}_x) = c_1 \tag{10.35}$$

where c_1 is another constant related to c_0. As $\mathbf{\Sigma}^{-1}$ is positive definite as well as $\mathbf{\Sigma}$, this equation represents an hyper ellipsoid in the d-dimensional space. In particular, when $d = 2$, $\mathbf{x} = [x_1,\ x_2]^T$, with positive definite $\mathbf{\Sigma}_x^{-1}$:

$$\mathbf{\Sigma}_x^{-1} = \begin{bmatrix} A & B/2 \\ B/2 & C \end{bmatrix} \qquad \text{and} \qquad \left|\mathbf{\Sigma}_x^{-1}\right| = AC - B^2/4 > 0 \tag{10.36}$$

then the quadratic equation above becomes

$$(\mathbf{x}-\mathbf{m}_x)^T\mathbf{\Sigma}_x^{-1}(\mathbf{x}-\mathbf{m}_x) = [x_1-\mu_{x_1}, x_2-\mu_{x_2}]\begin{bmatrix} A & B/2 \\ B/2 & C \end{bmatrix}\begin{bmatrix} x_1-\mu_{x_1} \\ x_2-\mu_{x_2} \end{bmatrix}$$
$$= A(x_1-\mu_{x_1})^2 + B(x_1-\mu_{x_1})(x_2-\mu_{x_2}) + C(x_2-\mu_{x_2})^2 = c_1$$

representing an ellipse (instead of other quadratic curves such as hyperbola and parabola) centered at $\mathbf{m}_x = [\mu_1,\ \mu_2]^T$. When $d = 3$, the quadratic equation represents an ellipsoid. In general when $d > 3$, the equation $\mathcal{N}(\mathbf{x}, \mathbf{m}_x, \mathbf{\Sigma}_x) = c_0$ represents a hyper-ellipsoid in the d-dimensional space.

Substituting $\mathbf{x} = \mathbf{V}\mathbf{y}$ and $\mathbf{m}_x = \mathbf{V}\mathbf{m}_y$ into the iso-surface equation Eq. (10.35), we get the equation for the hyper-ellipsoid after the KLT:

$$\begin{aligned}&(\mathbf{x}-\mathbf{m}_x)^T\mathbf{\Sigma}_x^{-1}(\mathbf{x}-\mathbf{m}_x) = [\mathbf{V}(\mathbf{y}-\mathbf{m}_y)]^T\mathbf{\Sigma}_x^{-1}\mathbf{V}\mathbf{y}-\mathbf{m}_y) \\ &= (\mathbf{y}-\mathbf{m}_y)^T\mathbf{V}^T\mathbf{\Sigma}_x^{-1}\mathbf{V}(\mathbf{y}-\mathbf{m}_y) = (\mathbf{y}-\mathbf{m}_y)^T\mathbf{\Lambda}^{-1}(\mathbf{y}-\mathbf{m}_y) \\ &= \sum_{i=1}^{d}\frac{(y_i-\mu_{y_i})^2}{\lambda_i} = \sum_{i=1}^{d}\frac{(y_i-\mu_{y_i})^2}{\sigma_{y_i}^2} = c_1\end{aligned} \tag{10.37}$$

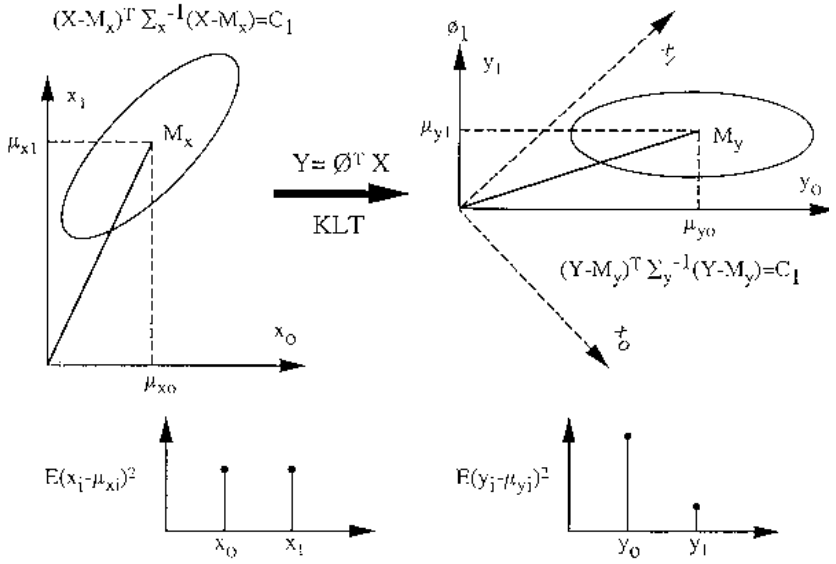


Figure 10.3 Standardization of ellipsoids by the KLT

This a standardized hyper-ellipsoid. We therefore see that the KLT transform $\mathbf{y} = \mathbf{V}^T\mathbf{x}$ is actually a rotation of the coordinate system of the d-dimensional space, which is spanned by the standard basis $\{\mathbf{e}_1, \cdots, \mathbf{e}_d\}$ before the KLT and the eigenvectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_d\}$ as the basis vectors after the KLT.

As the result, the principal semi-axes of the ellipsoid representing the Gaussian distribution of the dataset become in parallel with the axes of the new coordinate system, i.e., the ellipsoid becomes standardized. Moreover, the length of the ith principal semi-axis is proportional to the standard deviation $\sigma_{y_i} = \sqrt{\lambda_i}$ of the ith variable y_i. This is the reason why KLT possesses the two desirable properties: (a) the decorrelation of the signal components, and (b) redistribution and compaction of the energy or information contained in the signal, as illustrated in Fig. 10.3.

Example 10.1 The top row of Fig. 10.4 shows three given datasets with the following parameters:

$$\mathbf{\Sigma}_1 = \begin{bmatrix} 2.964 & 0.544 \\ 0.544 & 4.687 \end{bmatrix}, \qquad \mathbf{\Sigma}_2 = \begin{bmatrix} 5.036 & 3.083 \\ 3.083 & 2.830 \end{bmatrix}, \qquad \mathbf{\Sigma}_3 = \begin{bmatrix} 3.043 & -3.076 \\ -3.076 & 3.930 \end{bmatrix} \tag{10.38}$$

$$\rho_1 = 0.146, \qquad \rho_2 = 0.817, \qquad \rho_3 = -0.889 \tag{10.39}$$

Then the KLT is carried out to rotate the coordinate system from the orig-

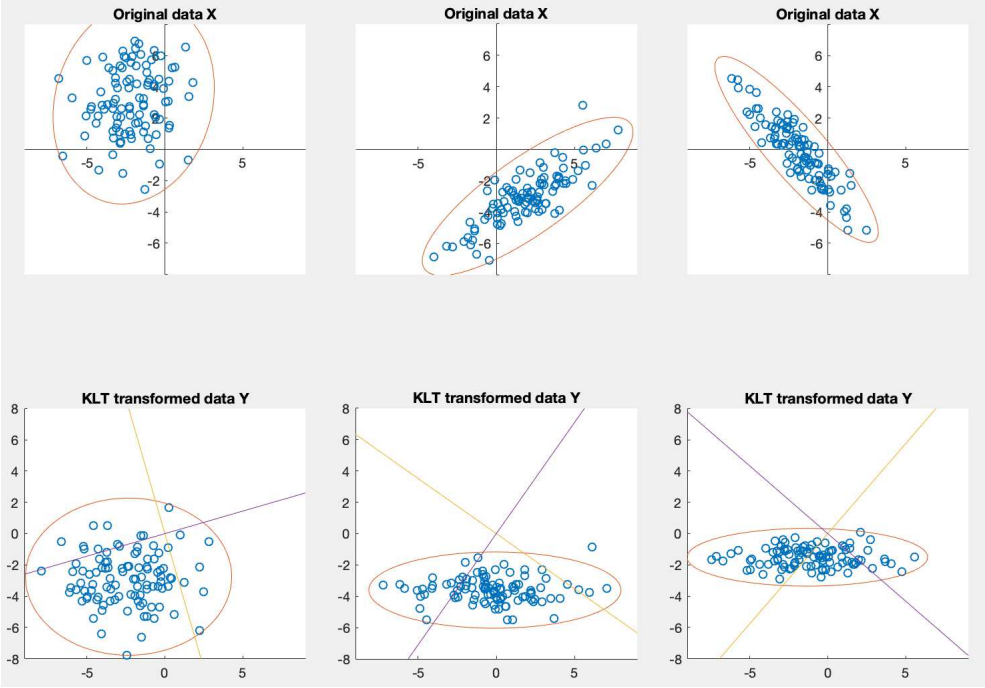


Figure 10.4 Three datasets before and after the KLT

inal standard basis to the eigenvectors of their covariance matrices, with the corresponding eigenvalues shown below:

$$\lambda_{1,2} = (4.845,\ 2.807), \qquad \lambda_{1,2} = (7.207,\ 0.659), \qquad \lambda_{1,2} = (6.594,\ 0.379) \tag{10.40}$$

The datasets after the KLT are shown in the bottom row of the figure. We see that in each of the three cases, the ellipse representing the distribution of the data points becomes standardized, with its semi-axes proportional to the eigenvalues, which, when squared, are the variances of the principal components.

10.5 Computation of the KLT

Due to KLT's properties of signal decorrelation and energy compaction, it can be used for data compression by reducing the dimensionality of the dataset. Specifically, we carry out the following steps:

1. Find the mean vector $\mathbf{m}_x$ and the covariance matrix $\mathbf{\Sigma}_x$ of the random vectors $\mathbf{x}$ based on the given dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ containing N samples of $\mathbf{x}$. Note that the rank of $\mathbf{\Sigma}$ is no greater than $N - 1$.
2. Find the eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$ of $\mathbf{\Sigma}_x$, sorted in descending order, and their corresponding eigenvectors $\{\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_d\}$.

3. Determine the reduced dimensionality $d' < d$ so that the percentage of energy contained after the transform is no less than a given threshold (e.g., 95%):

$$\frac{\sum_{i=1}^{d'} \lambda_i}{\sum_{i=1}^{d} \lambda_i} = \frac{\sum_{i=1}^{d'} \sigma_{y_i}^2}{\sum_{i=1}^{d} \sigma_{y_i}^2} \geq 0.95 \tag{10.41}$$

4. Construct a $d \times d'$ transform matrix composed of the d eigenvectors corresponding to the d largest eigenvalues of $\mathbf{\Sigma}_x$:

$$\mathbf{V}' = [\mathbf{v}_1, \cdots, \mathbf{v}_{d'}]_{d \times d'} \tag{10.42}$$

and carry out KLT based on $\mathbf{V}'$:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_{d'} \end{bmatrix}_{d' \times 1} = \mathbf{V}'^T \mathbf{x} = \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_{d'}^T \end{bmatrix}_{d' \times d} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_d \end{bmatrix}_{d \times 1} \tag{10.43}$$

This is a lossy compression by which the dimensionality is reduced from d to $d' < d$, with the following error representing the percentage of information lost:

$$\sum_{i=d'+1}^{d} \lambda_i / \sum_{i=1}^{d} \lambda_i \tag{10.44}$$

But as $\lambda_{d'+1}, \cdots, \lambda_d$ are the smallest $d - d'$ eigenvalues, the error is minimum (e.g., 5%).

5. Reconstruct $\mathbf{x}$ by inverse KLT:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_d \end{bmatrix}_{d \times 1} = \mathbf{V}'\mathbf{y} = \begin{bmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_{d'} \end{bmatrix}_{d \times d'} \begin{bmatrix} y_1 \\ \vdots \\ y_{d'} \end{bmatrix}_{d' \times 1} = \sum_{i=1}^{d'} y_i \mathbf{v}_i \tag{10.45}$$

The rank of the $d \times d$ estimated covariance matrix $\hat{\mathbf{\Sigma}}_x$ in Eq. (10.4) based on the dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ is no greater than $N - 1$. When $N \leq d$, then the rank of $\hat{\mathbf{\Sigma}}$ is at most $N - 1 < d$, i.e., there are no more than $N - 1$ non-zero eigenvalues while all remaining $d - (N - 1) = d - N + 1$ eigenvalues are zero. In this case, this estimated covariance matrix $\hat{\mathbf{\Sigma}}_x$ is not invertible.

Further more, if the dimensionality $d \gg N$ is much higher than the number of data samples, the computational complexity $O(d^3)$ for solving this eigenvalue problem of $\hat{\mathbf{\Sigma}}_x$ may be expensive. In this case, we can find the $N - 1$ non-zero eigenvalues and their corresponding eigenvectors of $\hat{\mathbf{\Sigma}}_x$ by solving the eigenvalue problem of an $N \times N$ matrix with much reduced complexity of $O(N^3)$.

Specifically, for convenience and without loss of generality, we assume that the

data have zero mean $\mathbf{m}_x = \mathbf{0}$, so that the covariance matrix of the given dataset $\mathbf{X}$ can be estimate as

$$\hat{\mathbf{\Sigma}}_x = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \mathbf{m}_x)(\mathbf{x}_n - \mathbf{m}_x)^T = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^T$$

$$= \frac{1}{N}[\mathbf{x}_1, \cdots, \mathbf{x}_N]\begin{bmatrix}\mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T\end{bmatrix} = \frac{1}{N}\mathbf{X}\mathbf{X}^T \tag{10.46}$$

and the eigenequation of this $d \times d$ matrix can be written as

$$\hat{\mathbf{\Sigma}}_x\mathbf{v}_i = \frac{1}{N}\mathbf{X}\mathbf{X}^T\mathbf{v}_i = \lambda_i\mathbf{v}_i \tag{10.47}$$

Pre-multiplying $\mathbf{X}^T$ on both sides, we get the eigenequation of an $N \times N$ matrix $\mathbf{X}^T\mathbf{X}/N$:

$$\frac{1}{N}\mathbf{X}^T\mathbf{X}\left(\mathbf{X}^T\mathbf{v}_i\right) = \lambda_i\left(\mathbf{X}^T\mathbf{v}_i\right) \qquad \text{i.e.} \qquad \frac{1}{N}\mathbf{X}^T\mathbf{X}\mathbf{u}_i = \lambda_i\mathbf{u}_i \tag{10.48}$$

where $\mathbf{u}_i = \mathbf{X}^T\mathbf{v}_i$ is ith eigenvector of $\mathbf{X}^T\mathbf{X}/N$. We see that $\mathbf{X}\mathbf{X}^T/N$ and $\mathbf{X}^T\mathbf{X}/N$ share the same non-zero eigenvalues. When $N < d$, we can solve the eigenvalue problem in Eq. (10.48) with complexity $O(N^3)$, instead that of the original problem in Eq. (10.47) with $O(d^3)$, to get the $N-1$ non-zero eigenvalues λ_i, $(i = 1, \cdots, N-1)$, and the corresponding eigenvectors $\mathbf{u}_i = \mathbf{X}^T\mathbf{v}_i$. To find the eigenvectors $\mathbf{v}_i$ of the original matrix $\hat{\mathbf{\Sigma}}_x = \mathbf{X}\mathbf{X}^T/N$, we pre-multiply $\mathbf{X}$ on both sides of the eigenequation in Eq. (10.48) and get

$$\frac{1}{N}\mathbf{X}\mathbf{X}^T\left(\mathbf{X}\mathbf{u}_i\right) = \lambda_i\left(\mathbf{X}\mathbf{u}_i\right) \tag{10.49}$$

Comparing this with Eq. (10.47), we see that $\mathbf{X}\mathbf{u}_i = \mathbf{v}_i$.

and they become the same when normalized.

10.6 Comparison with Other Orthogonal Transforms

Here we compare the KLT, in terms of signal decorrelation and energy compaction, with other orthogonal transforms such as discrete cosine transform (DCT) and Fourier transform (DFT), as well as identity transform IT (no transform), based on two images of cloud and sand as shown below.

We treat each column of the image as an observation (instantiation) of a 1-D random vector $\mathbf{x}$, and apply an orthogonal transforms $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ to $\mathbf{x}$ based on the covariance matrix $\mathbf{\Sigma}_\mathbf{x}$, and compare the corresponding covariance matrices $\mathbf{\Sigma}_y$ after the transform to see how well each transform decorrelates the signal and compacts its energy.

Fig. 10.5 shows two images (left) and the corresponding covariance matrices in image form (pixel values proportional to the values in the covariance matrix) after three orthogonal transforms of IT (no transform), DCT, and KLT.

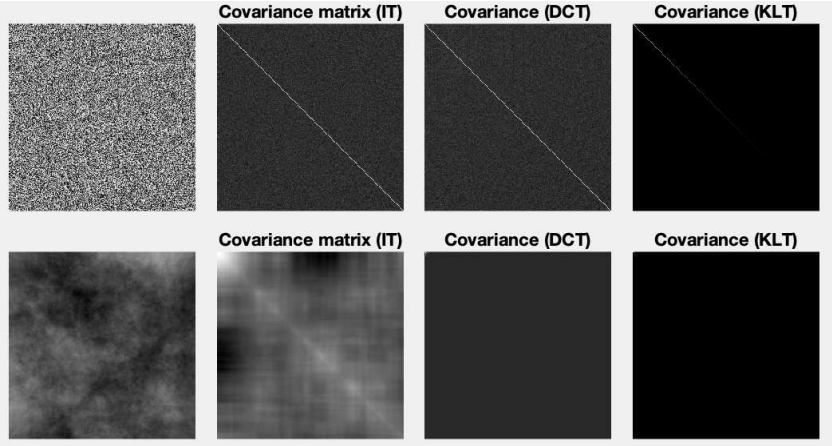


Figure 10.5 Images of Clouds and Sands

- Sand image

  In the second panel showing the covariance after IT, the off-diagonal pixels are dark, indicating the pixels in the original image are not highly correlated at all. In the third panel, the covariance after DCT looks similar to that before the transform, indicating that the DCT has little effect in terms of signal decorrelation and energy compaction. Finally, in the right-most panel showing the covariance after KLT, all off diagonal elements are zero, i.e., the signal components are completely decorrelated.

- Cloud image

  In the second panel for the covariance after IT, there exist some bright areas off the diagonal, indicating that many signal components close to each other are highly correlated. In the third panel for the covariance after DCT, the values of the off diagonal elements are significantly reduced, indicating that the signal components are significantly decorrelated. Finally, in the right-most panel showing the covariance after KLT, all off diagonal elements are zero, i.e., the signal components are completely decorrelated.

The effect of energy redistribution and compaction of these different transforms are also shown in Fig. 10.6, where the variances of signal components are sorted and plotted.

We see that due to the physical nature of the clouds and sand grains, the textures in the corresponding images are very different. In the cloud image, the neighboring pixels are highly correlated, while in the sand image, they are not much correlated at all. Correspondingly, the DCT can effectively decorrelate the cloud image, but much less so in the sand image. But in either case, whether the pixels in the image are highly correlated or not, the KLT can always completely decorrelate the signal and optimally compact the energy.

The table below further demonstrate the effect of energy compaction quantita-

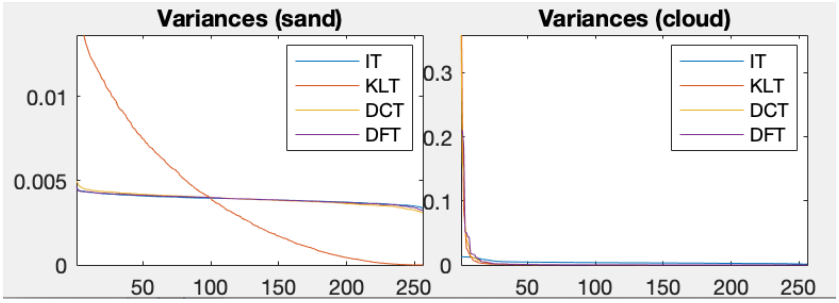


Figure 10.6 Energy compaction and redistribution by the KLT

tively in terms of the number of components out of the total of 256 components needed after the transform in order to keep a certain percentage of the total dynamic energy (information). For example, if it is desired to keep 95% of the total energy contained in the original signal, 230 components are needed with no transform, 22 are needed after DCT, and only 13 are needed after KLT. Note that DCT's performance is reasonably close to that of the optimal KLT in this case.

Percentage:	90%	95%	99%	100%
IT:	209	230	250	256
DCT:	10	22	97	256
KLT:	7	13	55	256

(10.50)

Although KLT is optimal in terms of signal decorrelation and energy compaction, it is not as convenient as other transforms for two reasons. First, the KLT is data-specific, i.e., the transform matrix is the eigenvector matrix of the covariance matrix of the dataset, which needs to be estimated based on sufficient amount of data samples, while all other orthogonal transforms are data independent. Second, the computational cost of the KLT is significantly higher than that of other transforms, due to the $O(d^3)$ complexity for solving the eigenvalue problem of the $d \times d$ matrix covariance matrix $\mathbf{\Sigma}_x$, while for most other popular orthogonal transforms fast algorithm exist with $O(d \log_2 d)$ complexity instead of $O(d^2)$ required by the KLT.

10.7 Application to Image Data

The KLT can be applied to a stack of N images each containing K pixels, as shown on the left of Fig. 10.7. Depending on the specific need (such as data compression and feature extraction), the KLT can be carried out in either of two alternative manners, which define the random signal vector $\mathbf{x}$ differently, as shown below.

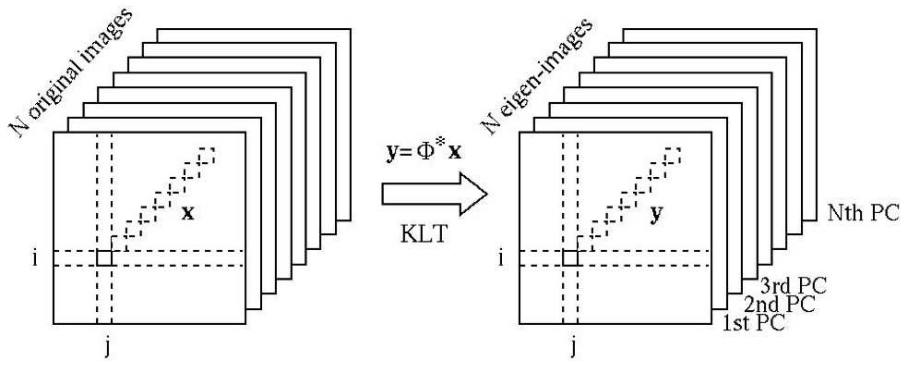


Figure 10.7 The KLT of a set of images

- A K-dimensional vector $\mathbf{x}_n$ $(n = 1, \cdots, N)$ can be formed by all K pixels of each of the N images by concatenating the rows (or columns) of the image array (same as what we did for the sand and cloud images). All N such vectors are treated as the column vectors of a $K \times N$ data array $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$. The $K \times K$ covariance matrix of these K-D column vectors can be estimated (assuming with zero mean) as:

$$\boldsymbol{\Sigma} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^T = \frac{1}{N}[\mathbf{x}_1, \cdots, \mathbf{x}_N]\begin{bmatrix}\mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T\end{bmatrix} = \frac{1}{N}\left(\mathbf{X}\mathbf{X}^T\right)_{K\times K} \quad (10.51)$$

- An N-dimensional vector can be formed by the pixels at the same position (e.g., in the ith row and jth column of the image array) from all N images. There are K such vectors each for one pixel in the images, and they are actually the rows of matrix $\mathbf{X}$ defined above, or the columns of $\mathbf{X}^T$. The $N \times N$ covariance matrix of these N-D row vectors can be estimated as:

$$\boldsymbol{\Sigma}' = \frac{1}{K}\left(\mathbf{X}^T\mathbf{X}\right)_{N\times N}. \quad (10.52)$$

As discussed previously, these two different covariance matrices share the same eigenvalues. The eigenequations for $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\mathbf{X}$ (with the constant coefficients $1/K$ and $1/N$ neglected) are:

$$\mathbf{X}^T\mathbf{X}\mathbf{v} = \lambda\mathbf{v}, \qquad \mathbf{X}\mathbf{X}^T\mathbf{u} = \mu\mathbf{u}. \quad (10.53)$$

Pre-multiplying $\mathbf{X}^T$ on both sides of the second equation we get

$$\mathbf{X}^T\mathbf{X}[\mathbf{X}^T\mathbf{u}] = \mu[\mathbf{X}^T\mathbf{u}]. \quad (10.54)$$

which is actually the first eigenequation with the same eigenvalue $\mu = \lambda$ and eigenvector $\mathbf{v} = \mathbf{X}^T\mathbf{u}$ when normalized. The two covariance matrices $\boldsymbol{\Sigma}$ and $\boldsymbol{\Sigma}'$ have the same rank $R = \min(N, K)$ (if $\mathbf{X}$ is not degenerate) and therefore the

same number of non-zero eigenvalues. Consequently, the KLT can be carried out based on either matrix with the same effects in terms of the signal decorrelation and energy compaction. As the number of pixels in the image is typically much greater than the number of images, $K > N$, we will take the second approach above to treat the pixels in the same position in all N images as a sample of an N-dimensional random signal vector and carry out the KLT based on the $N \times N$ covariance matrix $\hat{\mathbf{\Sigma}}$.

We can now carry out the KLT to each of the K N-dimensional vectors $\mathbf{x}$ corresponding to each pixel of the N images to obtain another N-dimensional vector $\mathbf{y} = \mathbf{v}^*\mathbf{x}$ for the same pixel of a set of N *eigenimages*, as shown on the right of Fig. 10.7. After the KLT, most of the energy/information contained in the N images, representing the signal variations among all N images, is concentrated in the first few eigenimages corresponding to the greatest eigenvalues, while the remaining eigenimages can be omitted without losing much energy/information. This is the foundation for various KLT-based image compression and feature extraction algorithms. The subsequent operations such as image recognition and classification can all be carried out in a much lower dimensional space.

We now consider some of such applications.

- **Video image compression**

  A sequence of $N = 8$ frames of a video of a moving escalator and their eigenimages are shown respectively in the upper and lower parts of Fig. 10.8.

  It is interesting to observe that the first eigenimage corresponding to the greatest eigenvalue (left panel of the third row of the figure) represents mostly the static scene of the image frames representing the main variations in the image (carrying most of the energy), while the subsequent eigenimages represent mostly the motion in the video, the variation between the frames. For example, the motion of the people riding on the escalator is mostly reflected by the first few eigenimages following the first one, while the motion of the escalator stairs is mostly reflected in the subsequent eigenimages.

  The 8×8 covariance matrix and the energy distribution among the eight components plot before and after the KLT are shown in Fig. 10.9

  We see that due to the spatial correlation between nearby pixels, the covariance matrix before the KLT (left) can be modeled by the squared exponential function, while the covariance matrix after the KLT (middle) is completely decorrelated and the energy is highly compacted into a small number of principal components (here the first component), as also clearly shown in the comparison of the energy distribution before and after the KLT (right).

- **Eigen-face and face recognition**

  The KLT is applied to a stack of twenty face images ($N = 20$) shown in Fig. 10.10 (credit to AT&T Laboratories Cambridge, downloadable from

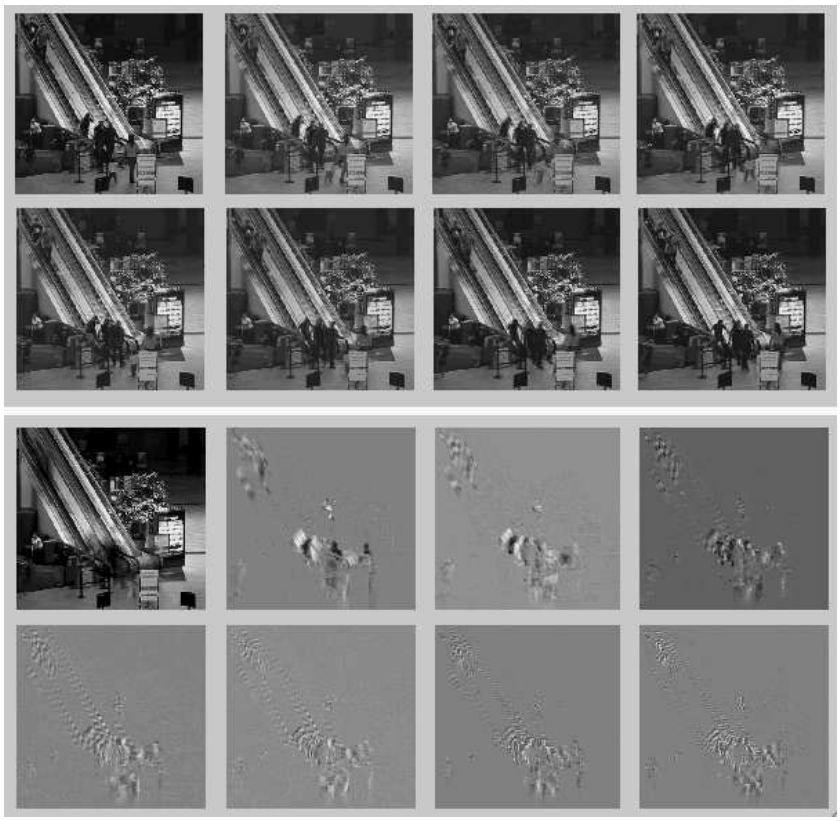


Figure 10.8 Eight frames of a video before (top) and after (bottom) the KLT

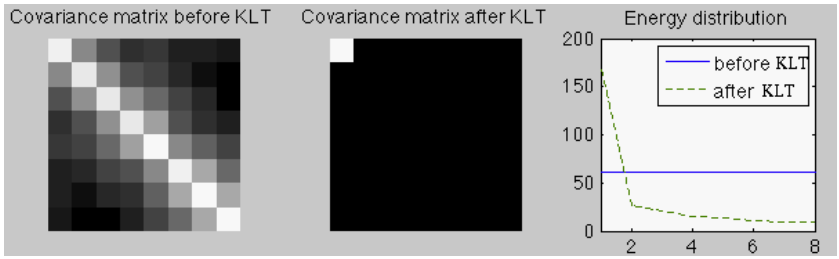


Figure 10.9 Covariance matrix before (left) and after (middle) the KLT, and energy redistribution (right)

https://cam-orl.co.uk/facedatabase.html), to generate the *eigen-faces* in Fig. 10.11.

The table below shows the percentage of energy contained in each of the



Figure 10.10 Face images

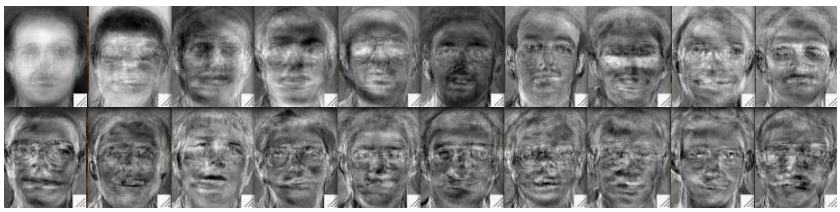


Figure 10.11 The eigen-faces

components:

components	1	2	3	4	5	6	7	8	9
percentage energy	48.5	11.6	6.1	4.6	3.8	3.7	2.6	2.5	1.9
accumulative energy	48.5	60.1	66.2	70.8	74.6	78.3	81.0	83.5	85.4

10	11	12	13	14	15	16	17	18	19	20
1.9	1.8	1.6	1.5	1.4	1.3	1.2	1.1	1.1	0.9	0.8
87.3	89.	90.7	92.2	93.6	94.9	96.1	97.2	98.2	99.2	100.0

(10.55)

The faces are then reconstructed based on 95% of the total information (15 out of 20 components) as shown in Fig. 10.12.

- **Hand-written digit recognition**

  The goal here is to recognize hand-written digits from 0 to 9 given in image form of 16×16 pixels, in a dataset containing $N = 224 \times 10$ samples for the $K = 10$ digits. A small subset of the samples is shown in Fig. 10.13. All $16 \times 16 = 256$ pixels in the image of each sample can be converted into a $d = 256$ vector by concatenating all columns (or rows) of the image. Then

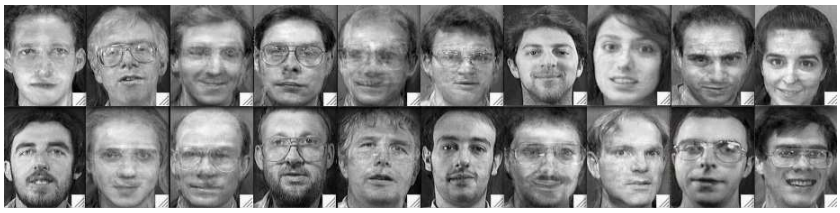


Figure 10.12 Reconstructed faces

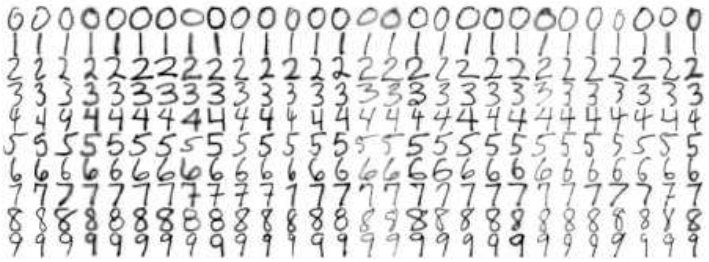


Figure 10.13 Ten handwritten digits

the KLT can be carried out to reduce the dimensionality from $d = 256$ to some $d' \ll d$, based on either the covariance matrix $\mathbf{\Sigma}_x = \mathbf{S}_T$ of all $N = 2240$ sample vectors representing the overall distribution of the dataset, or the between-class scatter matrix $\mathbf{S}_B$ previously considered representing the separability of the ten classes.

Specifically, we use the eigenvectors corresponding to the d' greatest eigenvalues of the covariance matrix or the between-class scatter matrix to form a d' by d transform matrix. After the KLT transform by the data, certain classification algorithm can be carried out in the much reduced d' dimensional space.

The energy distribution over all $d = 256$ signal components is plotted in Fig. 10.14 for both the original signal before the KLT (top) and after the KLT based on both $\mathbf{S}_B$ (middle) and $\mathbf{S}_T$ (bottom).

For the KLT based on $\mathbf{S}_T$ with rank $N - 1$, $d' = 79$ components are needed to keep 95.1% of the total energy, in comparison to the KTL based on $\mathbf{S}_B$ with rank $\text{rank}(\mathbf{S}_B) = K - 1 = 9$, requiring are only $d' = 9$ principal components corresponding to the same number of non-zero eigenvalues of $\mathbf{S}_B$ to keep 100% of the total energy representing the separability information. The percentage energy contained in these non-zero eigenvalues are: 28.6%, 22.7%, 14.4%, 10.9%, 7.1%, 7.0%, 3.8%, 3.2%, 2.4%.

The $d = 256$ dimensional eigenvectors can be visualized when converted into 16×16 eigen-digits, representing the basis by which any original face images can be represented as a linear combination of such eigen-digits as shown in Fig. 10.15. We see that the 10th eigen-digit corresponding to a zero eigenvalue contains no meaningful information but random noise.

If we only keep the first two or three principal components (corresponding to the greatest eigenvalues) after the KLT, the dataset can be visualized as shown in Fig. 10.16. The sample points in each of the ten different classes are color-coded. It can be seen that even when the dimensions are much

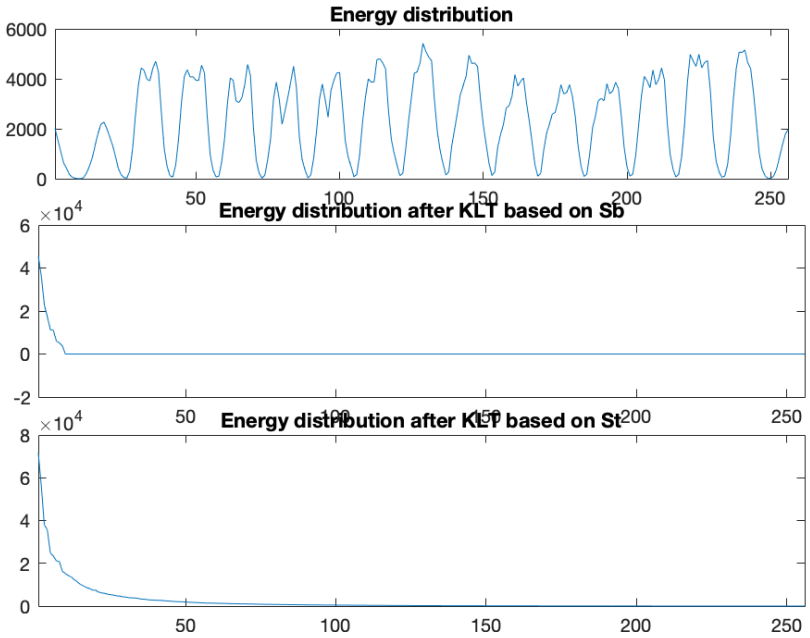


Figure 10.14 Energy distribution before (top) and after the KLT based on $\mathbf{S}_B$ (middle), and $\mathbf{S}_T$ (bottom)

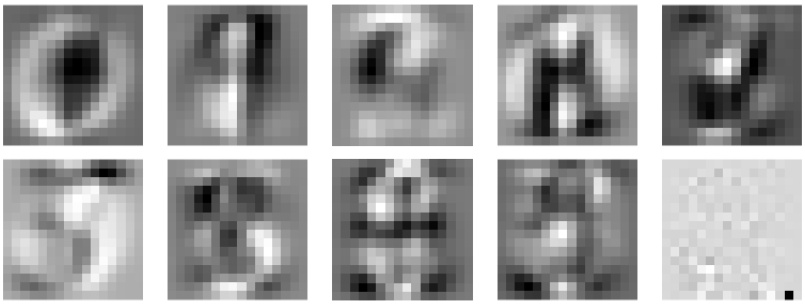


Figure 10.15 The first ten Eigen-digits

reduced from $N = 256$ to $d = 3$ or even $d = 2$, it is still possible to separate the ten classes reasonably well.

10.8 PCA for Feature Extraction

IN the PCA method considered in previous sections, the transform matrix is the eigenvector matrix of the covariance matrix of the dataset, of which the diagonal

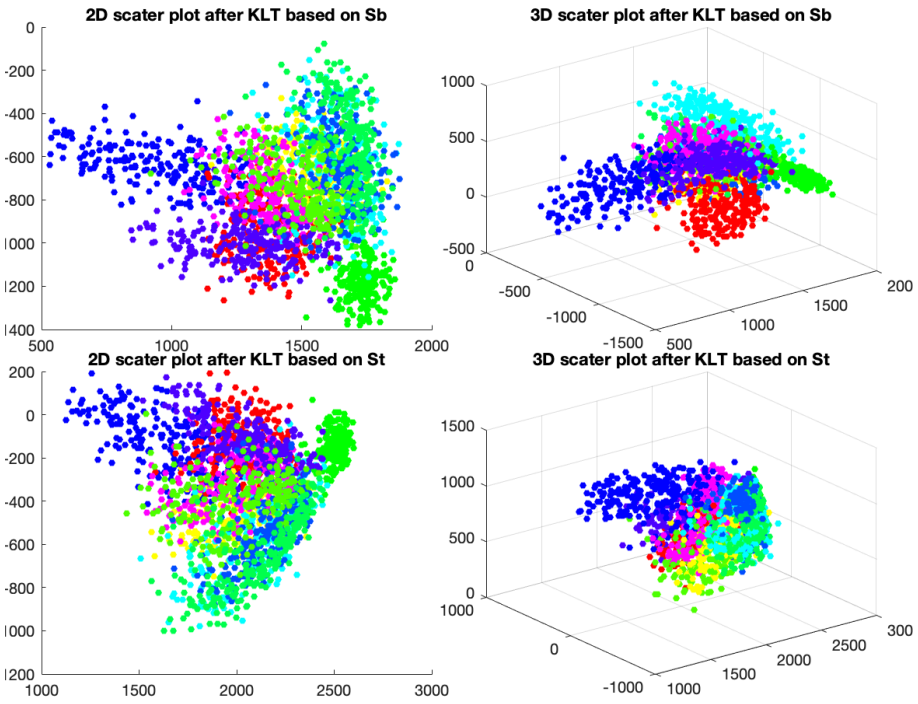


Figure 10.16 Visualization of ten classes of digits

components are the variances of the signal components that measure the dynamic energy or information contained in all data points in the entire dataset.

In the context of classification, we can also use the PCA method to reduce the data dimensionality while maximally preserving the information of interest, which in this case should be pertinent to the separability of data points into different classes, instead of just the generic variation in all data points independent of their classes. Specifically, the transform matrix $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_{d'}]$ in the linear transformation

$$\mathbf{y} = \mathbf{A}^T \mathbf{x} \tag{10.56}$$

needs to be constructed based on certain measurement of the class separability, such as the within and between scatter matrices considered in Chapter 9, instead of the covariance matrix of the entire dataset.

- **Optimal transformation for maximizing** $\mathrm{tr}(\mathbf{S}_B)$

  We first consider the objective function $J_B(\mathbf{A})$ in the new d'-D space as

a function of the transform matrix $\mathbf{A}$:

$$J(\mathbf{A}) = \text{tr}\,(\mathbf{S}_B^y) = \text{tr}\,(\mathbf{A}^T\,\mathbf{S}_B^x\,\mathbf{A}) = \text{tr}\begin{bmatrix}\mathbf{a}_1^T\\ \vdots\\ \mathbf{a}_{d'}^T\end{bmatrix}\mathbf{S}_B^x[\mathbf{a}_1,\cdots,\mathbf{a}_{d'}]$$

$$= \text{tr}\begin{bmatrix}\mathbf{a}_1^T\\ \vdots\\ \mathbf{a}_{d'}^T\end{bmatrix}[\mathbf{S}_B^x\mathbf{a}_1,\cdots,\mathbf{S}_B^x\mathbf{a}_{d'}] = \sum_{i=1}^{d'}\mathbf{a}_i^T\mathbf{S}_B^x\mathbf{a}_i \tag{10.57}$$

To find optimal $\mathbf{A}$ that maximizes $\text{tr}(\mathbf{S}_B^y)$, we need to solve the following constrained maximization problem:

$$\begin{cases}\text{maximize} & J(\mathbf{A}) = \text{tr}(\mathbf{A}^T\mathbf{S}_B^x\mathbf{A}) = \sum_{i=1}^{d'}\mathbf{a}_i^T\mathbf{S}_B^x\mathbf{a}_i\\ \text{subject to} & ||\mathbf{a}_j||^2 = \mathbf{a}_j^T\mathbf{a}_j = 1 \quad (j=1,\cdots,d')\end{cases} \tag{10.58}$$

where the constraint equations are to guarantee that the column vectors of $\mathbf{A}$ are normalized and therefore finite. This constrained optimization problem can be solved by Lagrange multiplier method. We first construct the Lagrangian function

$$L(\mathbf{A},\boldsymbol{\lambda}) = J(\mathbf{A}) - \sum_{j=1}^{d'}\lambda_j(\mathbf{a}_j^T\mathbf{a}_j - 1) \tag{10.59}$$

and then set its gradient with respect to each column vector $\mathbf{a}_i$ of $\mathbf{A}$ to zero:

$$\frac{\partial}{\partial\mathbf{a}_i}L(\mathbf{A},\boldsymbol{\lambda}) = \frac{\partial}{\partial\mathbf{a}_i}\left[\sum_{j=1}^{d'}(\mathbf{a}_j^T\mathbf{S}_B^x\mathbf{a}_j - \lambda_j\mathbf{a}_j^T\mathbf{a}_j + \lambda_j)\right]$$

$$= \frac{\partial}{\partial\mathbf{a}_i}\left[\mathbf{a}_i^T\mathbf{S}_B^x\mathbf{a}_i - \lambda_i\mathbf{a}_i^T\mathbf{a}_i\right] = 2\mathbf{S}_B^x\mathbf{a}_i - 2\lambda_i\mathbf{a}_i = 0, \quad \text{i.e.} \quad \mathbf{S}_B\mathbf{a}_i = \lambda_i\mathbf{a}_i$$
$$(i=1,\cdots,n) \tag{10.60}$$

This happens to be the ith eigenequation of $\mathbf{S}_B$, indicating the column vectors of the optimal $\mathbf{A}$ have to be the orthogonal eigenvectors $\{\mathbf{v}_1,\cdots,\mathbf{v}_{d'}\}$ of the symmetric matrix $\mathbf{S}_B^x$, i.e., $\mathbf{A} = [\mathbf{a}_1,\cdots,\mathbf{a}_{d'}] = \mathbf{V} = [\mathbf{v}_1,\cdots,\mathbf{v}_{d'}]$. Now the d' new features can be obtained as

$$\mathbf{y} = \mathbf{A}^T\mathbf{x} = \begin{bmatrix}\mathbf{v}_1^T\\ \vdots\\ \mathbf{v}_{d'}^T\end{bmatrix}_{d'\times d}\mathbf{x}_{d\times 1}, \quad \text{or} \quad y_i = \mathbf{v}_i^T\mathbf{x}, \ (i=1,\cdots,d') \tag{10.61}$$

and

$$J(\mathbf{A}) = \sum_{i=1}^{d'}\mathbf{v}_i^T\mathbf{S}_B\mathbf{v}_i = \sum_{i=1}^{d'}\mathbf{v}_i^T\mathbf{v}_i\lambda_i = \sum_{i=1}^{d'}\lambda_i > 0 \tag{10.62}$$

We note that as $\mathbf{S}_B^x$ is symmetric and positive definite, its eigenvalues are

positive and its normalized eigenvectors are orthonormal: $\mathbf{v}_i^T\mathbf{v}_j = \delta_{ij}$. To maximize $J(\mathbf{A})$, the eigenvectors in $\mathbf{A}$ need to correspond to the d' greatest eigenvalues of $\mathbf{S}_B$.

- **Optimal transformation for maximizing** $\text{tr}(\mathbf{S}_{B/T})$

  The previous method only maximizes $\mathbf{S}_B$ without taking $\mathbf{S}_W$ or $\mathbf{S}_T = \mathbf{S}_B + \mathbf{S}_W$ into consideration. As $\mathbf{S}_W^y$ in the d'-D space may have also changed as well as $\mathbf{S}_B^y$. To maximize the separability, we need to maximize $\mathbf{S}_B$ while at the same time also minimize $\mathbf{S}_W$. Or, equivalently, we need to maximize $\mathbf{S}_B^y$ normalized by $\mathbf{S}_T^y$:

$$\begin{aligned}\mathbf{S}_{B/T}^y &= (\mathbf{S}_T^y)^{-1}\,\mathbf{S}_B^y = (\mathbf{A}^T\mathbf{S}_T^x\mathbf{A})^{-1}(\mathbf{A}^T\mathbf{S}_B^x\mathbf{A}) \\ &= \mathbf{A}^{-1}(\mathbf{S}_T^x)^{-1}(\mathbf{A}^T)^{-1}\mathbf{A}^T\mathbf{S}_B^x\mathbf{A} \\ &= \mathbf{A}^{-1}(\mathbf{S}_T^x)^{-1}\mathbf{S}_B^x\mathbf{A} = \mathbf{A}^{-1}\mathbf{S}_{B/T}^x\mathbf{A} \end{aligned} \tag{10.63}$$

  We note that different from the symmetric matrix $\mathbf{S}_B$ considered above, matrix $\mathbf{S}_{B/T}^y$ as a product of two symmetric matrices is no longer symmetric, i.e., its eigenvectors are not orthogonal in general, and the method above cannot be used, we will find the optimal matrix $\mathbf{A}$ in some other way.

  We first consider the special case of $d' = 1$, i.e., $\mathbf{A} = \mathbf{a}$ is an $d \times 1$ vector and $y = \mathbf{a}^T\mathbf{x}$ is a scalar. We desire to maximize the following objective function in a 1-D space:

$$(\mathbf{S}_T^y)^{-1}\,\mathbf{S}_B^y = (\mathbf{a}^T\mathbf{S}_T^x\mathbf{a})^{-1}(\mathbf{a}^T\mathbf{S}_B^x\mathbf{a}) = \frac{s_B^y}{s_T^y} = \frac{\mathbf{a}^T\mathbf{S}_B^x\mathbf{a}}{\mathbf{a}^T\mathbf{S}_T^x\mathbf{a}} = R(\mathbf{a}) \tag{10.64}$$

  This function $R(\mathbf{a})$ is the *Rayleigh quotient* (Section A.3.3) of the two symmetric matrices $\mathbf{S}_B^x$ and $\mathbf{S}_T^x$. The optimal transform vector $\mathbf{a}$ that maximizes $R(\mathbf{a})$ can be found by solving the corresponding *generalized eigenvalue problem* (Section A.3.2):

$$\mathbf{S}_B^x\mathbf{v}_i = \lambda_i\mathbf{S}_T^x\mathbf{v}_i, \qquad (i = 1, \cdots, d) \tag{10.65}$$

  where $\lambda_i = R(\mathbf{a})$ is an eigenvalue and $\mathbf{v}_i$ the corresponding eigenvector of $\mathbf{S}_{B/T} = (\mathbf{S}_T^x)^{-1}\mathbf{S}_B^x$. Obviously, the transform vector $\mathbf{a}$ that maximizes $R(\mathbf{a})$ is the eigenvector corresponding to the greatest eigenvalue $\lambda_{\max} \geq \lambda_i$ $(i = 1, \cdots, d)$.

  We next generalize the method above to the case of $d' > 1$. The matrix form of the eigenequation above is:

$$\mathbf{S}_B^x\mathbf{V} = \mathbf{S}_T^x\mathbf{V}\mathbf{\Lambda} \tag{10.66}$$

  where $\mathbf{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_d)$ is the diagonal eigenvalue matrix of $(\mathbf{S}_T^x)^{-1}\mathbf{S}_B^x$, and $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_d]$ the eigenvector matrix which is no longer orthogonal in general. This generalized eigenvalue problem can be solved by finding the matrix $\mathbf{V}$ that diagonalizes both $\mathbf{S}_T^x$ and $\mathbf{S}_B^x$ at the same time (same

as how to get Eq. (A.137)):

$$\begin{cases} \mathbf{S}_B^y = \mathbf{V}^T \mathbf{S}_B^x \mathbf{V} = \mathbf{\Lambda} \\ \mathbf{S}_T^y = \mathbf{V}^T \mathbf{S}_T^x \mathbf{V} = \mathbf{I} \end{cases} \tag{10.67}$$

Left multiplying $(\mathbf{S}_T^y)^{-1} = \mathbf{I}$ to the first, we get

$$\mathbf{S}_{B/T}^y = (\mathbf{S}_T^y)^{-1} \mathbf{S}_B^y = \mathbf{\Lambda} \tag{10.68}$$

The optimal transform matrix is composed of the eigenvectors $\mathbf{A} = [\mathbf{v}_1, \cdots, \mathbf{v}_{d'}]$ corresponding to the d' greatest of all d eigenvalues $\lambda_1 \geq \cdots \geq \lambda_{d'} \geq \cdots \geq \lambda_d$, so that

- the signal components in $\mathbf{y} = [y_1, \cdots, y_{d'}]^T$ are completely decorrelated, each component $y_i = \mathbf{v}_i^T \mathbf{x}$ carries certain separability information $s_B^{(y_i)} / s_T^{(y_i)} = \lambda_i$ $(i = 1, \cdots, d')$, independent of others;
- the total separability contained in the d'-D space, as the sum of the d' greatest eigenvalues $\sum_{i=1}^{d'} \lambda_i$, is maximized.

Problems

In this problem set and some later ones as well as many examples in future chapters, we will again use the handwritten digit dataset considered in Section 10.7. A small subset of the dataset is shown in Fig. 10.13, while the complete dataset containing 2240 handwritten digits can be downloaded from the Cambridge University Press website at www.[to come].org.

Also, an ASCII file containing a 256×2240 array $\mathbf{X}$ for all 2240 data samples in vector form and a 256-D vector $\mathbf{y}$ for their class labelings can be downloaded here: /e176/homework/data0to9.txt

This ASCII file can be read by the code below:

```
data=load('data0to9.txt','data');  $ read ASCII file
d=size(data,1)-1        % dimensionality of data
N=size(data,2);         % total number of samples
X=data(1:d,:);          % all N d-dimensional sample vectors
y=data(d+1,:);          % and their labelings
K=length(unique(y));    % number of classes
for k=1:K               % for each of the K classes
    idx=find(y==k);     % indices of points in class k
    Xk=X(:,idx)         % collect all points in class k
end
```

1. Carry out KLT for the 4-D iris dataset. Display the 4×4 covariance matrices before and after the transform, and verify (1) the trace (sum of all diagonal components of the variance matrix) is conserved, (2) all off-diagonal components become zero after the KLT, i.e., all signal components are completely decorrelated.

2. Visualize the 4-D iris dataset in both 2-D and 3-D spaces spanned respectively by the first two and three principal components. Color-code all 150 data points as red, green, or blue according to their class identities.
3. Carry out KLT for the handwritten digit dataset and observe the effects of signal decorrelation and energy redistribution of the transform. Display the 256×256 covariance matrices both before and after the transform in image form, same as in Figs. 10.5 and 10.9, and plot their diagonal components (representing the dynamic energy or information contained in each data component), same as in Fig. 10.14.
4. Visualize the handwritten digit dataset in both 2-D and 3-D spaces spanned respectively by the first two and three principal components. Color code all data points according to their class identities to appreciate how difficult it is to separate them into different classes. (That is what you need to do in the future.)
5. Find all pair-wise Bhattacharyya distances $d_B(C_i, C_j),\ i, i = 2, \cdots, 10,\ j = 1, \cdots, i-1\}$ between any two of the ten classes in the handwritten digits dataset, and display them in a lower triangular distance matrix.

   Note:
   - As the 256-D covariance matrix $\mathbf{\Sigma}_k$ is estimated based on only 224 samples of class C_k, the rank of this estimated covariance matrix is no more than $224 - 1 = 223$.
   - As all 224 samples are for the same digit, they are similar to each other and highly correlated, the rank of $\mathbf{\Sigma}_k$ is likely to be much lower than 223.

   We therefore see that the estimated $\mathbf{\Sigma}_k$ is non-invertible as its rank is much smaller than its dimension, and so is the sum of two such covariance matrices (needed in the Bhattacharyya distance). Consequently, the dimensionality of the dataset must be reduced significantly ($d' \ll d = 256$) to become smaller than the rank of $\mathbf{\Sigma}_k$ by the KLT transform, before the Bhattacharyya distances can be calculated.
6. Typically the KLT matrix is calculated as the eigenvector matrix of the covariance matrix $\mathbf{\Sigma}$ (same as the total scatter matrix $\mathbf{S}_T$) of the entire dataset, and the resulting principal components reflect the signal variations of all data points in the dataset. Alternatively, the KLT can also be carried out based on the between-class scatter matrix $\mathbf{S}_B$ or $\mathbf{S}_{B/T}$ to maximize the trace (or equivalently based on the within-class scatter matrix $\mathbf{S}_W$ to minimize its trace), if the signal variation of interest is the separability of different classes in the context of classification.

   Now carry out such a KLT to reduce the dimensionality of both the iris and handwritten digit datasets based on $\mathbf{S}_B$. Visualize the datasets in both the 2-D and 3-D spaces spanned by the greatest principal components, with all data points color-coded according to their class identities.
7. As discussed in section 10.7, the method of PCA can be applied to image data based on the covariance matrices given in Eq. (10.7). Now carry out PCA to

the face images, downloadable from https://cam-orl.co.uk/facedatabase.html). Observe the resulting eigen-faces to see how different types of variations among all faces are extracted by the eigen-faces corresponding to the first few greatest eigenvalues.

8. Repeat the previous problem based on the images of handwritten numbers. Observe the resulting eigen-digits.

11 Variations of PCA

In this chapter we consider a set of methods that are either some variations of the PCA method in the previous chapter or closely related to it to achieve similar effects for feature selection and extraction.

11.1 Kernel Methods

In the kernel method, all data points as a vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ in the original d-dimensional feature space are mapped by a *kernel function* $\mathbf{z} = \phi(\mathbf{x})$ into a higher, possibly infinite, dimensional space

$$\mathbf{x} \Longrightarrow \mathbf{z} = \phi(\mathbf{x}) \tag{11.1}$$

The presumption of the method is that the data points only appear in the form of inner product $\mathbf{x}_m^T \mathbf{x}_n$ in the algorithm, then based on the *kernel trick*, the kernel function $\mathbf{z} = \phi(\mathbf{x})$ never needs to be actually carried out. In fact, the form of the kernel function $\phi(\mathbf{x})$ and the dimensionality of the higher dimensional space do not need to be explicitly specified or known.

The motivation for such a kernel mapping is that the relevant operations such as classification and clustering may be carried out much more effectively once the dataset is mapped to the higher dimensional space. For example, classes not linearly separable in the original d-dimensional feature space may be trivially separable in a higher dimensional space, as illustrated by the following examples.

Example 11.1 In 1-D space, two classes $C_- = \{x \big| (a \leq x \leq b)\}$ and $C_+ = \{x \big| (x \leq a) \text{ or } (x \geq b)\}$ shown in the first row of Fig. 11.1 are not linearly separable. However, by the following mapping from 1-D space to 2-D space:

$$\mathbf{z} = \phi(x) = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} x \\ (x - (a+b)/2)^2 \end{bmatrix} \tag{11.2}$$

the two classes can be separated by thresholding in the second dimension of the 2-D space.

Example 11.2 The method above can be generalized to spaces of higher dimensions. For example, the two classes in 2-D space shown in the middle row of Fig.

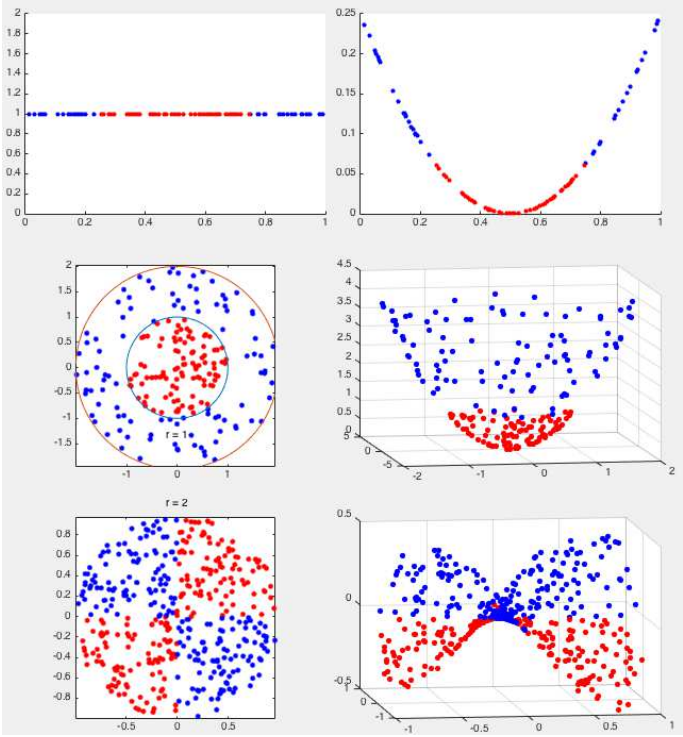


Figure 11.1 Examples of kernel mapping

11.1 are not linearly separable: $C_- = \{\mathbf{x},\ ||\mathbf{x}|| < D\}$ and $C_+ = \{\mathbf{x},\ ||\mathbf{x}|| > D\}$. However, by the following mapping from 2-D space to 3-D space:

$$\mathbf{z} = \phi(\mathbf{x}) = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 + x_2^2 \end{bmatrix} \tag{11.3}$$

the two classes can be trivially separated linearly by thresholding in the third dimension of the 3-D space.

Example 11.3 In 2-D space, in the exclusive OR dataset shown in the bottom row of Fig. 11.1 the two classes of C_- containing points in quadrants I and III, and C_+ containing points in quadrants II and IV are not linearly separable. However, by mapping the data points to a 3-D space:

$$\mathbf{z} = \phi(\mathbf{x}) = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix} \tag{11.4}$$

the two classes can be separated by simply thresholding in the third dimension of the 3-D space.

Definition: A kernel is a function that takes two vectors $\mathbf{x}_m$ and $\mathbf{x}_n$ as arguments and returns the inner product of their images $\mathbf{z}_m = \phi(\mathbf{x}_m)$ and $\mathbf{z}_n = \phi(\mathbf{x}_n)$:

$$K(\mathbf{x}_m, \mathbf{x}_n) = \phi(\mathbf{x}_m)^T \phi(\mathbf{x}_n) = \mathbf{z}_m^T \mathbf{z}_n \tag{11.5}$$

The kernel function takes as input some two vectors $\mathbf{x}_m$ and $\mathbf{x}_n$ in the original space, and returns a scalar value as the inner product $\mathbf{z}_m$ and $\mathbf{z}_n$ in some higher dimensional space. If the data points in the original space only appear in the form of inner product, then the kernel function $\mathbf{z} = \phi(\mathbf{x})$, called the kernel-induced *implicit* mapping, never needs to be explicitly specified, and the dimension of the new does not even need to be known.

The following is a set of commonly used kernel functions $K(\mathbf{x}, \mathbf{x}')$, which can be represented as an inner product of two vectors $\mathbf{z} = \phi(\mathbf{x})$ and $\mathbf{z}' = \phi(\mathbf{x}')$ in a higher dimensional space.

- **linear kernel**

  Given two vectors $\mathbf{x}$ and $\mathbf{x}'$, the linear kernel is defined as the true inner product, without any kernel mapping:

$$K(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}' = \sum_{i=1}^{d} x_i x_i' \tag{11.6}$$

- **polynomial kernels**

  The binomial theorem states:

$$(x+y)^n = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) x^{n-k} y^k = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} x^{n-k} y^k \tag{11.7}$$

  where the binomial coefficient

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!} \tag{11.8}$$

  is the number of ways to distribute n items into two bins (k in one and $n-k$ in the other). This result can be generalized to the multinomial case:

$$\begin{aligned} (x_1 + \cdots + x_d)^n &= \sum_{\sum_{i=1}^{d} k_i = n} \left(\begin{array}{c} n \\ k_1, \cdots, k_d \end{array} \right) x_1^{k_1} \cdots x_d^{k_d} \\ &= \sum_{\sum_{i=1}^{d} k_i = n} \frac{n!}{k_1! \cdots k_d!} x_1^{k_1} \cdots x_d^{k_d} \end{aligned} \tag{11.9}$$

  where the multinomial coefficient

$$\left(\begin{array}{c} n \\ k_1, \cdots, k_d \end{array} \right) = \frac{n!}{k_1! \cdots k_d!} \tag{11.10}$$

  is the number of ways to distribute n balls into d bins with k_i balls in the ith bin, and the summation is over all possible ways to get d non-negative integers $k_1, \cdots, k_d$ that add up to n.

Now consider the homogeneous polynomial kernel for d-dimensional vectors $\mathbf{x} = [x_1, \cdots, x_d]^T$ defined as

$$\begin{aligned} K(\mathbf{x}, \mathbf{x}') &= (\mathbf{x}^T\mathbf{x}')^n = (x_1x_1' + \cdots + x_dx_d')^n \\ &= \sum_{\sum_{i=1}^d k_i = n} \frac{n!}{k_1! \cdots k_d!} \left((x_1x_1')^{k_1} \cdots (x_dx_d')^{k_d}\right) \\ &= \phi(\mathbf{x})^T\phi(\mathbf{x}') = \mathbf{z}^T\mathbf{z}' \end{aligned} \tag{11.11}$$

where

$$\mathbf{z} = \phi(\mathbf{x}) = \left[\sqrt{\frac{n!}{k_1! \cdots k_d!}} \left(x_1^{k_1} \cdots x_d^{k_d}\right), \left(k_i \geq 0, \sum_{i=1}^d k_i = n\right)\right]^T \tag{11.12}$$

In particular, when $d = 2$ and $n = 2$, the polynomial kernel defined over 2-D vectors $\mathbf{x} = [x_1, x_2]^T$ is:

$$\begin{aligned} K(\mathbf{x}, \mathbf{x}') &= (\mathbf{x}^T\mathbf{x}')^2 = (x_1x_1' + x_2x_2')^2 \\ &= (x_1x_1')^2 + 2x_1x_1'x_2x_2' + (x_2x_2')^2 = \phi(\mathbf{x})^T\phi(\mathbf{x}') = \mathbf{z}^T\mathbf{z}' \end{aligned} \tag{11.13}$$

where $\mathbf{z} = \phi(\mathbf{x}) = [x_1^2, \sqrt{2}x_1x_2, x_2^2]$ is a mapping from $\mathbf{x}$ in 2-D space to $\mathbf{z}$ in 3-D space.

A non-homogeneous polynomial kernel is defined as

$$K(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T\mathbf{x}')^n \tag{11.14}$$

- **The radial basis function (RBF) kernel**

  The RBF kernel is defined as

$$K(\mathbf{x}, \mathbf{x}') = e^{-||\mathbf{x}-\mathbf{x}'||^2/2\sigma^2} = e^{-\gamma||\mathbf{x}-\mathbf{x}'||^2} \tag{11.15}$$

  where $\gamma = 1/2\sigma^2$ is a parameter that can be adjusted to fit each specific dataset. This kernel can be written as the inner product of two infinite dimensional vectors (for simplicity, we assume $\sigma = 1$):

$$\begin{aligned} K(\mathbf{x}, \mathbf{x}') &= e^{-||\mathbf{x}-\mathbf{x}'||^2/2} = e^{-||\mathbf{x}||^2/2}\, e^{-||\mathbf{x}'||^2/2}\, e^{\mathbf{x}^T\mathbf{x}'} = e^{-||\mathbf{x}||^2/2}\, e^{-||\mathbf{x}'||^2/2} \sum_{n=0}^{\infty} \frac{(\mathbf{x}^T\mathbf{x}')^n}{n!} \\ &= e^{-||\mathbf{x}||^2/2}\, e^{-||\mathbf{x}'||^2/2} \sum_{n=0}^{\infty} \left[\frac{1}{n!} \sum_{\sum_{i=1}^d k_i = n} \frac{n!}{k_1! \cdots k_d!} \left((x_1x_1')^{k_1} \cdots (x_dx_d')^{k_d}\right)\right] \\ &= \sum_{n=0}^{\infty} \sum_{\sum_{i=1}^d k_i = n} \left(e^{-||\mathbf{x}||^2/2} \frac{x_1^{k_1} \cdots x_d^{k_d}}{\sqrt{k_1! \cdots k_d!}}\right) \left(e^{-||\mathbf{x}'||^2/2} \frac{x_1'^{k_1} \cdots x_d'^{k_d}}{\sqrt{k_1! \cdots k_d!}}\right) \\ &= \phi(\mathbf{x})^T\phi(\mathbf{x}') = \mathbf{z}^T\mathbf{z}' \end{aligned} \tag{11.16}$$

  where

$$\mathbf{z} = \phi(\mathbf{x}) = \left[e^{-||\mathbf{x}||^2/2} \frac{x_1^{k_1} \cdots x_d^{k_d}}{\sqrt{k_1! \cdots k_d!}}, \left(n = 0, \cdots, \infty, \sum_{k=1}^n k_i = n\right)\right]^T \tag{11.17}$$

is a vector in an infinite dimensional space. In particular, when $d = 1$ we have

$$K(x,\, x') = e^{-(x-x')^2/2} = e^{-x^2/2}\, e^{-x'^2/2}\, e^{xx'} = e^{-x^2/2}\, e^{-x'^2/2} \sum_{n=0}^{\infty} \frac{(xx')^n}{n!}$$

$$= \sum_{n=0}^{\infty} (e^{-x^2/2}\, x^n/\sqrt{n!})\; (e^{-x'^2/2}\, x'^n/\sqrt{n!}) \tag{11.18}$$

where $\mathbf{z} = \phi(x) = \left[e^{-x^2/2}\, x^n/\sqrt{n!},\ (n = 0, \cdots, \infty)\right]^T$ is a kernel function that maps a 1-D space into an infinite dimensional space.

11.2 Kernel PCA

The PCA method maps the dataset to a lower dimensional space, and the kernel method maps the dataset to a higher dimensional space. These two methods are combined in kernel PCA to take advantage of separability in the high dimensional space and computational efficiency in the low dimensional space.

Specifically, we assume the given dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ in the original d-dimensional space is mapped into $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_N]$ in a high dimensional space. Correspondingly, the inner product $\mathbf{x}_m^T \mathbf{x}_n$ in the original space is mapped into an inner product in the high dimensional space by a kernel function:

$$\mathbf{z}_m^T \mathbf{z}_n = K(\mathbf{x}_m, \mathbf{x}_n) = k_{mn} \qquad (m, n = 1, \cdots, N) \tag{11.19}$$

The N by N kernel matrix can be found as:

$$\mathbf{K} = \mathbf{Z}^T\mathbf{Z} = \begin{bmatrix} \mathbf{z}_1^T \\ \vdots \\ \mathbf{z}_N^T \end{bmatrix} [\mathbf{z}_1, \cdots, \mathbf{z}_N] = \begin{bmatrix} \mathbf{z}_1^T\mathbf{z}_1 & \cdots & \mathbf{z}_1^T\mathbf{z}_N \\ \vdots & \ddots & \vdots \\ \mathbf{z}_N^T\mathbf{z}_1 & \cdots & \mathbf{z}_N^T\mathbf{z}_N \end{bmatrix} = \begin{bmatrix} k_{11} & \cdots & k_{1N} \\ \vdots & \ddots & \vdots \\ k_{N1} & \cdots & k_{NN} \end{bmatrix} \tag{11.20}$$

To carry out KLT in the high dimensional space, we need to estimate the covariance matrix of $\mathbf{z} = \phi(\mathbf{x})$:

$$\mathbf{\Sigma}_z = \frac{1}{N} \sum_{n=1}^{N} (\mathbf{z}_n - \mathbf{m}_z)(\mathbf{z}_n - \mathbf{m}_z)^T = \frac{1}{N} \sum_{n=1}^{N} \mathbf{z}_n \mathbf{z}_n^T - \mathbf{m}_z \mathbf{m}_z^T \tag{11.21}$$

If we assume the mean vector of all N samples in $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_N]$ is zero $\mathbf{m}_z = \mathbf{0}$ (to be justified later), $\mathbf{\Sigma}_z$ can be written as (same as Eq. (10.46)):

$$\mathbf{\Sigma}_z = \frac{1}{N} \sum_{n=1}^{N} (\mathbf{z}_n - \mathbf{m}_z)(\mathbf{z}_n - \mathbf{m}_z)^T = \frac{1}{N} \sum_{n=1}^{N} \mathbf{z}_n \mathbf{z}_n^T = \frac{1}{N} [\mathbf{z}_1, \cdots, \mathbf{z}_N] \begin{bmatrix} \mathbf{z}_1^T \\ \vdots \\ \mathbf{z}_N^T \end{bmatrix} = \frac{1}{N} \mathbf{Z}\mathbf{Z}^T \tag{11.22}$$

We further get its eigenequations:

$$\mathbf{\Sigma}_z \mathbf{v}_n = \left(\frac{1}{N}\mathbf{Z}\mathbf{Z}^T\right)\mathbf{v}_n = \lambda_n \mathbf{v}_n \qquad (n = 1, 2, \cdots) \tag{11.23}$$

To express this equations in terms of the kernel matrix $\mathbf{K} = \mathbf{K}^T\mathbf{K}$, we premultiply $\mathbf{Z}^T$ on both sides of the eigenequation above to get (same as Eq. (10.48)):

$$\frac{1}{N}\mathbf{Z}^T\mathbf{Z}\left(\mathbf{Z}^T\mathbf{v}_n\right) = \frac{1}{N}\mathbf{K}\mathbf{u}_n = \lambda_n\left(\mathbf{Z}^T\mathbf{v}_n\right) = \lambda_n\mathbf{u}_n \tag{11.24}$$

where we have defined $\mathbf{u}_n = \mathbf{Z}^T\mathbf{v}_n$. Solving this eigenequation of the $N \times N$ matrix $\mathbf{K}/N$ we get its eigenvectors $\mathbf{u}_n$ corresponding to the eigenvalues λ_n, the same as the non-zero eigenvalues of $\mathbf{\Sigma}_z = \mathbf{Z}\mathbf{Z}^T/N$. The eigenvectors $\mathbf{v}_n$ of $\mathbf{\Sigma}_z$ in Eq. (11.23) can now be written as:

$$\mathbf{v}_n = \frac{1}{\lambda_n N}\mathbf{Z}\left(\mathbf{Z}^T\mathbf{v}_n\right) = \frac{1}{\lambda_n N}\mathbf{Z}\mathbf{u}_n \tag{11.25}$$

which can be further normalized by imposing the condition $||\mathbf{v}_n||^2 = 1$:

$$\begin{aligned} ||\mathbf{v}_n||^2 = \mathbf{v}_n^T\mathbf{v}_n &= \frac{1}{(\lambda_n N)^2}(\mathbf{Z}\mathbf{u}_n)^T(\mathbf{Z}\mathbf{u}_n) = \frac{1}{(\lambda_n N)^2}\mathbf{u}_n^T\mathbf{Z}^T\mathbf{Z}\mathbf{u}_n \\ &= \frac{1}{(\lambda_n N)^2}\mathbf{u}_n^T\mathbf{K}\mathbf{u}_n = \frac{1}{\lambda_n N}\mathbf{u}_n^T\mathbf{u}_n = \frac{1}{\lambda_n N}||\mathbf{u}_n||^2 = 1 \end{aligned} \tag{11.26}$$

where we have used the fact $\mathbf{K}\mathbf{u}_n = \lambda_n N\mathbf{u}_n$ in Eq. (11.24). We can get the normalized eigenvectors of $\mathbf{\Sigma}_z$ in Eq. (11.23):

$$\mathbf{v}_n = \frac{1}{\sqrt{\lambda_n N}}\mathbf{u}_n \tag{11.27}$$

Now we can carry out KLT for any data point $\mathbf{z} = \phi(\mathbf{x})$ in the high dimensional space by

$$\mathbf{y} = \mathbf{V}^T\mathbf{z} = [\mathbf{v}_1, \cdots, \mathbf{v}_N]^T\mathbf{z} \tag{11.28}$$

of which each component $y_n = \mathbf{v}_n^T\mathbf{z}$ is the projection of $\mathbf{z} = \phi(\mathbf{x})$ onto one of the eigenvectors $\mathbf{v}_n$ of $\mathbf{\Sigma}_z$ $(n = 1, \cdots, N)$

$$\begin{aligned} y_n = \mathbf{z}^T\mathbf{v}_n = \mathbf{z}^T\left(\frac{1}{\lambda_n N}\mathbf{Z}\mathbf{u}_n\right) &= \frac{1}{\lambda_n N}\mathbf{z}^T[\mathbf{z}_1, \cdots, \mathbf{z}_N]\mathbf{u}_n \\ = \frac{1}{\lambda_n N}[\mathbf{z}^T\mathbf{z}_1, \cdots, \mathbf{z}^T\mathbf{z}_N]\,\mathbf{u}_n = \frac{1}{\lambda_n N}[k_1, \cdots, k_N]\,\mathbf{u}_n &= \frac{1}{\lambda_n N}\mathbf{k}^T\mathbf{u}_n \end{aligned} \tag{11.29}$$

where we have defined $\mathbf{k} = [k_1, \cdots, k_N]^T$ with $k_n = \mathbf{z}^T\mathbf{z}_n$. Although neither $\mathbf{z}$ nor $\mathbf{v}_n = \mathbf{Z}\mathbf{u}_n/\lambda_n N$ is explicitly available, their inner product $y_n = \mathbf{z}^T\mathbf{v}_n$ can still be obtained based on the kernel function $\mathbf{k}$ and $\mathbf{u}$. As alll data points in the high dimensional space appear only in the form of an inner product in the equation above, as well as in the eigenequation Eq. (11.24), the kernel mapping $\mathbf{z} = \phi(\mathbf{x})$ never needs to be explicitly carried out.

The discussion above is based on the assumption that in the high dimensional space the data have zero mean $\mathbf{m}_z = \mathbf{0}$. This cannot be achieved by subtracting

the mean from the data to get $\tilde{\mathbf{z}} = \mathbf{z} - \mathbf{m}_z$, as neither $\mathbf{z}$ nor $\mathbf{m}_z$ is available without actually carrying out the kernel mapping $\mathbf{z} = \phi(\mathbf{x})$. However, we can find the inner product of any two zero-mean data points $\tilde{\mathbf{z}}_m,\ \tilde{\mathbf{z}}_n,\ (m, n = 1, \cdots, N)$ in the high dimensional space:

$$\begin{aligned}\tilde{k}_{mn} = \tilde{\mathbf{z}}_m^T \tilde{\mathbf{z}}_n &= (\mathbf{z}_m - \mathbf{m}_z)^T (\mathbf{z}_n - \mathbf{m}_z) = \left(\mathbf{z}_m - \frac{1}{N}\sum_{k=1}^{N} \mathbf{z}_k\right)^T \left(\mathbf{z}_n - \frac{1}{N}\sum_{l=1}^{N} \mathbf{z}_l\right) \\ &= \mathbf{z}_m^T \mathbf{z}_n - \mathbf{z}_m^T \left(\frac{1}{N}\sum_{l=1}^{N} \mathbf{z}_l\right) - \left(\frac{1}{N}\sum_{k=1}^{N} \mathbf{z}_k^T\right) \mathbf{z}_n + \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{N} \mathbf{z}_k^T \mathbf{z}_l \\ &= k_{mn} - \frac{1}{N}\sum_{l=1}^{N} k_{ml} - \frac{1}{N}\sum_{k=1}^{N} k_{kn} + \frac{1}{N^2}\sum_{k=1}^{K}\sum_{l=1}^{N} k_{kl} \end{aligned} \tag{11.30}$$

The kernel matrix composed of all $N \times N$ such inner products can be written as:

$$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_N \mathbf{K} - \mathbf{K}\mathbf{1}_N + \mathbf{1}_N \mathbf{K} \mathbf{1}_N \tag{11.31}$$

where

$$\mathbf{1}_N = \frac{1}{N} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{N \times N} \tag{11.32}$$

In the algorithm considered above if we replace the kernel matrix $\mathbf{K}$ for $\mathbf{z}$ with non-zero mean by $\tilde{\mathbf{K}}$ for $\tilde{\mathbf{z}}$ with zero mean, then the assumption that $\mathbf{z}$ has zero mean $\mathbf{m}_z$ becomes valid. In summary, here are the steps of the kernel PCA algorithm:

- Given the dataset $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$, obtain $\mathbf{K}$ based on kernel mapping $k_{mn} = k(\mathbf{x}_m, \mathbf{x}_n)$ for all $m, n = 1, \cdots, N$, and then find $\tilde{\mathbf{K}}$, as in Eq. (11.31);
- Solve eigenvalue problem of $\tilde{\mathbf{K}}$ to get the eigenvalue λ_n and the corresponding eigenvector $\mathbf{u}_n$ for all $i = 1, \cdots, N$;
- Project each data point $\mathbf{z} = \phi(\mathbf{x})$ (implicit) onto the nth eigenvector $\mathbf{v}_n$ to get the principal component: $y_n = \mathbf{z}^T \mathbf{v}_n = \mathbf{k}^T \mathbf{u}_n / \lambda_n N$ for each $n = 1, \cdots, N$, as in Eq. (11.29).
- Carry out whatever operation (e.g., feature selection, classification) in the subspace spanned by the first $M < N$ principal components $\{y_1, \cdots, y_M\}$ in the high dimensional feature space, containing most of the information.

Note that in kernel PCA methods, the complexity for solving the eigenvalue problem of the $N \times N$ kernel matrix $\mathbf{K}$ is $O(N^3)$, instead of $O(d^3)$ for the linear PCA based on the $d \times d$ covariance matrix $\mathbf{\Sigma}_x$.

Shown below is a Metlab function for the kernel PCA algorithm which takes the dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ as input and generates the eigenvalues in $\mathbf{d}$ and

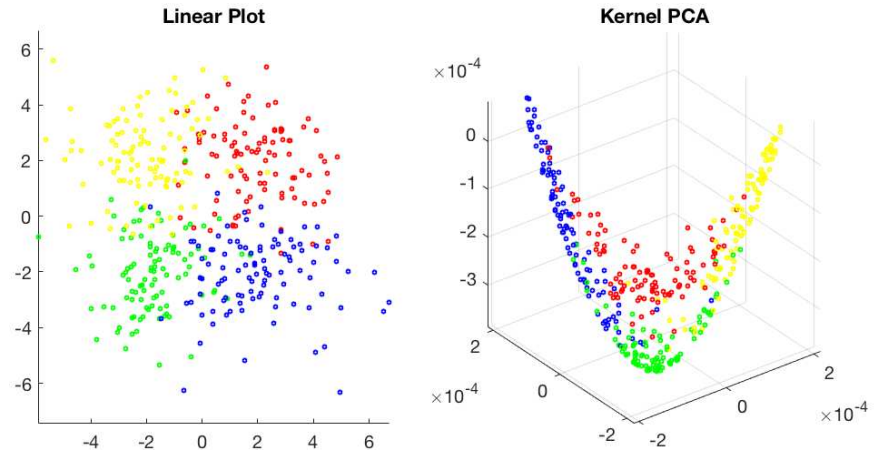


Figure 11.2 Linear and kernel PCAs for a 2-D dataset of four classes

the dataset in the transform space $\mathbf{Y}$. The function `Kernel(X,X)` calculates the kernel matrix $\mathbf{K}$.

```
function [Y d]=KPCA(X)
    [D N]=size(X);
    K=Kernel(X,X);                % kernel matrix of data X
    one=ones(N)/N;
    Kt=K-one*K-K*one+one*K*one;  % Kernel matrix of data with zero mean
    [U D]=eig(Kt);                % solve eigenvalue problem for K
    [d idx]=sort(diag(D),'descend');
                                  % sort eigenvalues in descending order
    U=U(:,idx);
    D=D(:,idx);
    for n=1:N
        U(:,n)=U(:,n)/d(n)/N;
    end
    Y=(K*U)';
end
```

The linear and kernel PCAs (based on RBF kernel) are applied to three datasets and compared as shown in Figs. 11.2 through 11.4.

11.3 Factor Analysis and Expectation Maximization

The method of *factor analysis (FA)* models a set of d observed *manifest variables* in $\mathbf{x} = [x_1, \cdots, x_d]^T$ as a linear combination of a set of $d' < d$ unobserved hidden *latent variables* or *common factors* in $\mathbf{z} = [z_1, \cdots, z_{d'}]^T$, to explain and reveal

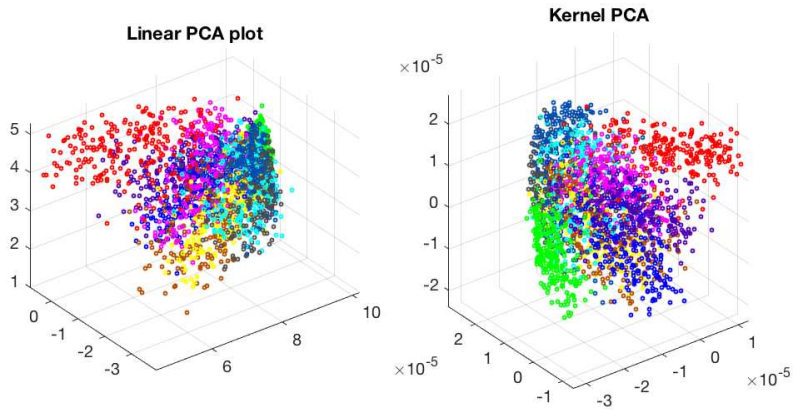


Figure 11.3 Linear and kernel PCAs for the 256-D handwritten digits

the variability and dependency among the d observed variables, typically correlated, in terms of the latent variables, assumed to be independent and therefore uncorrelated. The method of FA is therefore similar to the method of PCA in the sense that they both aim at extracting the essential information contained in a multivariate dataset as some linear combination of all signal components and thereby reducing the dimensionality of the dataset while preserving most of the information contained in it.

Specifically, we assume each of the observed variables in $\mathbf{x}$ is a linear combination of the d' factors in $\mathbf{z}$

$$x_i = \sum_{j=1}^{d'} w_{ij} z_j + e_i = [w_{i1}, \cdots, w_{id'}] \begin{bmatrix} z_1 \\ \vdots \\ z_{d'} \end{bmatrix} + e_i, \qquad (i = 1, \cdots, d) \quad (11.33)$$

or in matrix form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_d \end{bmatrix} = \begin{bmatrix} w_{11} & \cdots & w_{1d'} \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ w_{d1} & \cdots & w_{dd'} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_{d'} \end{bmatrix} + \begin{bmatrix} e_1 \\ \vdots \\ \vdots \\ e_d \end{bmatrix} = \mathbf{W}\mathbf{z} + \mathbf{e} \quad (11.34)$$

where $\mathbf{W}$ is a $d \times d'$ *factor loading matrix*, and e_i is the noise associated with x_i. Also, for simplicity and without loss of generality, we assume the dataset has a zero mean. If $\mathbf{W}$ were available, the d' factors in $\mathbf{z}$ can be found by solving this over-determined linear equation system of d equations but $d' < d$ unknowns by the least-squares method (with minimum squared error $||\mathbf{e}||^2$):

$$\hat{\mathbf{z}} = \mathbf{W}^{-}\mathbf{x} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{x} \quad (11.35)$$

where $\mathbf{W}^{-} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T$ is the left pseudo-inverse of $\mathbf{W}$.

However, as $\mathbf{W}$ is unavailable, it needs to be estimated together with $\mathbf{z}$ at

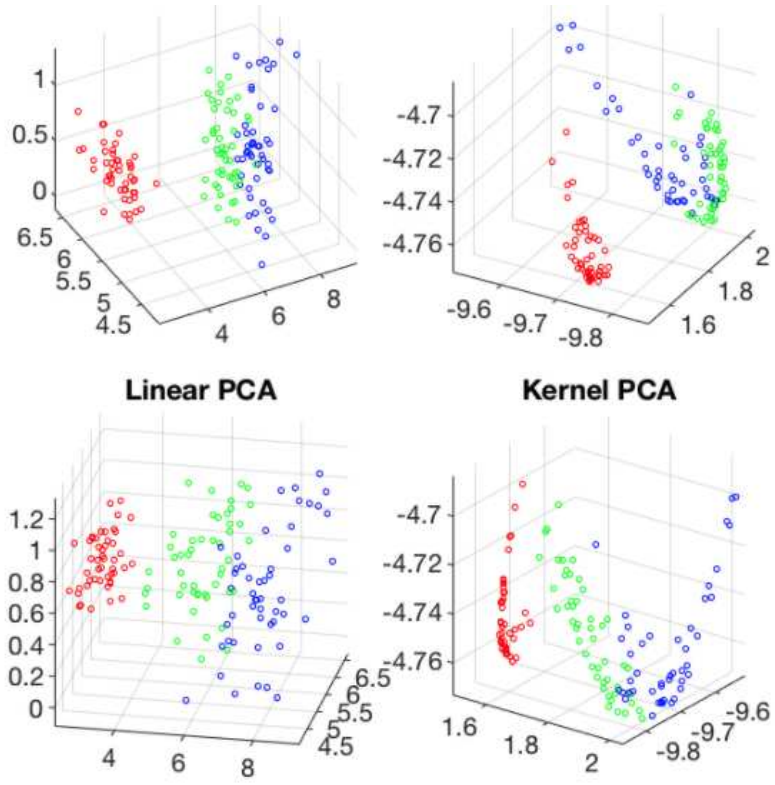


Figure 11.4 Linear and kernel PCAs for the iris dataset

the same time, based on the given dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, typically $N \gg d$. This can be done by the general method of *expectation-maximization (EM)*, as discussed below.

Specifically, we treat both $\mathbf{z}$ and $\mathbf{e}$ as random vectors, and make the following assumptions:

- The latent variables in $\mathbf{z}$ are of zero mean, independent of each other, and with unity variance:

$$\mathbf{m}_z = \mathrm{E}[\mathbf{z}] = \mathbf{0}, \qquad \mathbf{\Sigma}_z = \mathrm{Cov}[\mathbf{z}] = \mathrm{E}[\mathbf{z}\mathbf{z}^T] = \mathbf{I} \tag{11.36}$$

  and they are normally distributed:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{0},\ \mathbf{I}) \tag{11.37}$$

- The noise components in $\mathbf{e}$ have zero mean, and they are independent of each

other with a diagonal covariance matrix:

$$\mathbf{m}_e = \mathrm{E}[\mathbf{e}] = \mathbf{0}, \qquad \mathbf{\Sigma}_e = \mathrm{Cov}[\mathbf{e}] = \mathrm{E}[\mathbf{e}\mathbf{e}^T] = \mathbf{\Psi} = \begin{bmatrix} \psi_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \psi_d \end{bmatrix} \tag{11.38}$$

and they are also normally distributed:

$$p(\mathbf{e}) = \mathcal{N}(\mathbf{0}, \mathbf{\Psi}) \tag{11.39}$$

- The latent variables and the noise are independent of each other:

$$\mathbf{\Sigma}_{ze} = \mathrm{Cov}[\mathbf{z}, \mathbf{e}] = \mathrm{E}[(\mathbf{z} - \mathbf{m}_z)(\mathbf{e} - \mathbf{m}_e^T)] = \mathrm{E}[\mathbf{z}\mathbf{e}^T] = \mathbf{0} \tag{11.40}$$

The two matrices $\mathbf{W}$ and $\mathbf{\Psi}$ defined above as the parameters of the FA model are denoted by $\theta = \{\mathbf{W}, \mathbf{\Psi}\}$.

Based on the assumptions above, we desire to find the conditional pdf $p(\mathbf{z}|\mathbf{x})$ of the latent variable $\mathbf{z}$ given the observed variable $\mathbf{x}$. To do so, we first find the pdf of $\mathbf{x} = \mathbf{W}\mathbf{z} + \mathbf{e}$, which, as a linear combination of the two normally distributed random vectors $\mathbf{z}$ and $\mathbf{e}$, is also normally distributed with $p(\mathbf{x}) = \mathcal{N}(\mathbf{m}_x, \mathbf{\Sigma}_x)$, where

$$\mathbf{m}_x = \mathrm{E}[\mathbf{x}] = \mathrm{E}[\mathbf{W}\mathbf{z} + \mathbf{e}] = \mathbf{W}\,\mathrm{E}[\mathbf{z}) + \mathrm{E}(\mathbf{e}] = \mathbf{0} \tag{11.41}$$

$$\begin{aligned} \mathbf{\Sigma}_x = \mathrm{Cov}[\mathbf{x}] &= \mathrm{E}[(\mathbf{x} - \mathbf{m}_x)(\mathbf{x} - \mathbf{m}_x^T)] = \mathrm{E}[\mathbf{x}\mathbf{x}^T] = E\left[(\mathbf{W}\mathbf{z} + \mathbf{e})(\mathbf{W}\mathbf{z} + \mathbf{e})^T\right] \\ &= \mathbf{W}\,\mathrm{E}[\mathbf{z}\mathbf{z}^T]\mathbf{W}^T - \mathrm{E}[\mathbf{e}\mathbf{z}^T]\mathbf{W}^T - \mathbf{W}\,\mathrm{E}[\mathbf{z}\mathbf{e}^T] + \mathrm{E}[\mathbf{e}\mathbf{e}^T] \\ &= \mathbf{W}\mathbf{W}^T + \mathbf{\Psi} \end{aligned} \tag{11.42}$$

As both $\mathbf{m}_x$ and $\mathbf{\Sigma}_x$ are based on the model parameter $\theta = \{\mathbf{W}, \mathbf{\Psi}\}$, the pdf of $\mathbf{x}$ is conditional on θ:

$$p(\mathbf{x}|\theta) = \mathcal{N}(\mathbf{m}_x, \mathbf{\Sigma}_x) = \mathcal{N}(\mathbf{0}, \mathbf{W}\mathbf{W}^T + \mathbf{\Psi}) \tag{11.43}$$

The joint distribution $p(\mathbf{z}, \mathbf{x})$ is also Gaussian with a zero mean

$$\mathbf{m} = E\left(\begin{bmatrix} \mathbf{z} \\ \mathbf{x} \end{bmatrix}\right) = \begin{bmatrix} \mathbf{m}_z \\ \mathbf{m}_x \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{11.44}$$

and a covariance $\mathbf{\Sigma}$ composed of four submatrices:

$$\mathbf{\Sigma} = \begin{bmatrix} \mathbf{\Sigma}_{zz} & \mathbf{\Sigma}_{zx} \\ \mathbf{\Sigma}_{xz} & \mathbf{\Sigma}_{xx} \end{bmatrix} \tag{11.45}$$

where

$$\begin{aligned} \mathbf{\Sigma}_{zz} &= \mathbf{I} \\ \mathbf{\Sigma}_{xx} &= \mathbf{W}\mathbf{W}^T + \mathbf{\Psi} \\ \mathbf{\Sigma}_{zx} &= \mathbf{\Sigma}_{zx}^T = \mathrm{E}[(\mathbf{z} - \mathbf{m}_z)(\mathbf{x} - \mathbf{m}_x)^T] = \mathrm{E}[\mathbf{z}\mathbf{x}^T] = \mathrm{E}[\mathbf{z}(\mathbf{W}\mathbf{z} + \mathbf{e})^T] \\ &= \mathrm{E}[\mathbf{z}\mathbf{z}^T]\mathbf{W}^T + \mathrm{E}[\mathbf{z}\mathbf{e}^T] = \mathbf{I}\mathbf{W}^T + \mathbf{0} = \mathbf{W}^T \end{aligned} \tag{11.46}$$

The normal distribution $p(\mathbf{z}, \mathbf{x}|\theta)$ can now be expressed as:

$$p(\mathbf{x}, \mathbf{z}|\theta) = p\left(\begin{bmatrix} \mathbf{z} \\ \mathbf{x} \end{bmatrix}\right) = \mathcal{N}(\mathbf{m}, \boldsymbol{\Sigma}) = \mathcal{N}\left(\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{zz} & \boldsymbol{\Sigma}_{zx} \\ \boldsymbol{\Sigma}_{xz} & \boldsymbol{\Sigma}_{xx} \end{bmatrix}\right)$$
$$= \mathcal{N}\left(\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \mathbf{I} & \mathbf{W}^T \\ \mathbf{W} & \mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi} \end{bmatrix}\right) \tag{11.47}$$

Based on this joint pdf of both $\mathbf{x}$ and $\mathbf{z}$, we can further find the desired conditional pdfs of both $p(\mathbf{x}|\mathbf{z})$ and $p(\mathbf{z}|\mathbf{x})$ (Section B.1.5):

- $p(\mathbf{x}|\mathbf{z}, \theta) = \mathcal{N}(\mathbf{m}_{x|z}, \boldsymbol{\Sigma}_{x|z})$, with

$$\begin{cases} \mathbf{m}_{x|z} &= \mathbf{m}_x + \boldsymbol{\Sigma}_{xz}\boldsymbol{\Sigma}_{zz}^{-1}(\mathbf{z} - \mathbf{m}_z) = \mathbf{W}\mathbf{z} \\ \boldsymbol{\Sigma}_{x|z} &= \boldsymbol{\Sigma}_{xx} - \boldsymbol{\Sigma}_{xz}\boldsymbol{\Sigma}_{zz}^{-1}\boldsymbol{\Sigma}_{zx} = \mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi} - \mathbf{W}\mathbf{W}^T = \boldsymbol{\Psi} \end{cases} \tag{11.48}$$

- $p(\mathbf{z}|\mathbf{x}, \theta) = \mathcal{N}(\mathbf{m}_{z|x}, \boldsymbol{\Sigma}_{z|x})$, with

$$\begin{cases} \mathbf{m}_{z|x} &= \mathbf{m}_z + \boldsymbol{\Sigma}_{zx}\boldsymbol{\Sigma}_{xx}^{-1}(\mathbf{x} - \mathbf{m}_x) = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi})^{-1}\mathbf{x} = \mathbf{B}\mathbf{x} \\ \boldsymbol{\Sigma}_{z|x} &= \boldsymbol{\Sigma}_{zz} - \boldsymbol{\Sigma}_{zx}\boldsymbol{\Sigma}_{xx}^{-1}\boldsymbol{\Sigma}_{xz} = \mathbf{I} - \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi})^{-1}\mathbf{W} = \mathbf{I} - \mathbf{B}\mathbf{W} \end{cases} \tag{11.49}$$

where we have defined

$$\mathbf{B} = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi})^{-1} \tag{11.50}$$

Note that while $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ has zero mean and diagonal covariance, the conditional distribution $p(\mathbf{z}|\mathbf{x}, \theta) = \mathcal{N}(\mathbf{m}_{z|x}, \boldsymbol{\Sigma}_{z|x})$ has non-zero mean $\mathbf{m}_{z|x}$ and non-diagonal covariance $\boldsymbol{\Sigma}_{z|x}$.

The computational complexity for the inversion of the $N \times N$ matrix $\mathbf{W}\mathbf{W}^T + \boldsymbol{\Psi}$ is $O(N^3)$. However, by applying the Woodbury matrix identity (Section A.2.4):

$$(\boldsymbol{\Psi} + \mathbf{W}\mathbf{W}^T)^{-1} = \boldsymbol{\Psi}^{-1} - \boldsymbol{\Psi}^{-1}\mathbf{W}(\mathbf{I} + \mathbf{W}^T\boldsymbol{\Psi}^{-1}\mathbf{W})^{-1}\mathbf{W}^T\boldsymbol{\Psi}^{-1} \tag{11.51}$$

where $\boldsymbol{\Psi}$ as a diagonal matrix can be easily inverted, and $\mathbf{I} + \mathbf{W}^T\boldsymbol{\Psi}^{-1}\mathbf{W}$ is an $d' \times d'$ matrix that can be inverted with complexity $O(d'^3) \ll O(N^3)$.

The model parameter $\theta = \{\mathbf{W}, \boldsymbol{\Psi}\}$ can now be estimated based on the given dataset $\mathbf{X}$ by the EM algorithm, an iterative process of the following two steps:

- **The E-step:**

  Find the expectation of the log-likelihood function of the model parameter θ, to be maximized in the following M-step.

  Find the likelihood function of θ based on the observed dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ (all samples assumed to be i.i.d.):

$$L(\theta|\mathbf{X}, \mathbf{z}) = p(\mathbf{X}, \mathbf{z}|\theta) = \prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{z}|\theta) = \prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{z}, \theta)\, p(\mathbf{z}|\theta) \tag{11.52}$$

  and the log-likelihood:

$$\log L(\theta|\mathbf{X}, \mathbf{z}) = \sum_{n=1}^{N} (\log p(\mathbf{x}_n|\mathbf{z}, \theta) + \log p(\mathbf{z})) \tag{11.53}$$

The second term can be dropped as $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ is independent of the model parameter θ and therefore irrelevant to the maximization of the log-likelihood with respect to θ.

We then find the expectation of the log-likelihood function, denoted by Q, with respect to the latent variable $\mathbf{z}$ based on Eq. (11.48):

$$\begin{aligned}
Q &= E_{z|x}\left[\log L(\theta|\mathbf{X}, \mathbf{z})\right] = \sum_{n=1}^{N} E_{z|x_n}\left[\log p(\mathbf{x}_n|\mathbf{z}, \theta)\right] \\
&= \sum_{n=1}^{N} E_{z|x_n}\ \log\left[\frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}_{x|z}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}_n - \mathbf{m}_{x|z})^T \mathbf{\Sigma}_{x|z}^{-1}(\mathbf{x}_n - \mathbf{m}_{x|z})\right)\right] \\
&= \sum_{n=1}^{N} E_{z|x_n}\left[-\frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\mathbf{\Psi}| - \frac{1}{2}(\mathbf{x}_n - \mathbf{W}\mathbf{z})^T \mathbf{\Psi}^{-1}(\mathbf{x}_n - \mathbf{W}\mathbf{z})\right] \\
&= -\frac{dN}{2}\log(2\pi) - \frac{N}{2}\log|\mathbf{\Psi}| - \frac{1}{2}\sum_{n=1}^{N} E_{z|x_n}\left[(\mathbf{x}_n - \mathbf{W}\mathbf{z})^T \mathbf{\Psi}^{-1}(\mathbf{x}_n - \mathbf{W}\mathbf{z})\right] \\
&= C - \frac{N}{2}\log|\mathbf{\Psi}| - \frac{1}{2}\sum_{n=1}^{N} E_{z|x_n}\left[\mathbf{x}_n^T \mathbf{\Psi}^{-1}\mathbf{x}_n - 2\mathbf{x}_n^T \mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z} + \mathbf{z}^T \mathbf{W}^T \mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z}\right] \qquad (11.54)
\end{aligned}$$

where we have used the fact that

$$\mathbf{x}_n^T \mathbf{\Psi}^{-1}(\mathbf{W}\mathbf{z}) = (\mathbf{W}\mathbf{z})^T \mathbf{\Psi}^{-1}\mathbf{x}_n \qquad (11.55)$$

as both $\mathbf{\Psi}^{-1}$ and $\mathbf{\Psi}$ are diagonal and therefore also symmetric. Here the constant $C = -dN\log(2\pi)/2$ can be dropped.

- **The M-step:**

  Find the optimal model parameter $\theta = \{\mathbf{W}, \mathbf{\Psi}\}$ that maximizes the expectation of the log-likelihood Q obtained in the E-step.

  We set to zero the derivative of Q with respective each of the two parameters in $\theta = \{\mathbf{W}, \mathbf{\Psi}\}$ and solve the resulting equations.

  – Find $\mathbf{W}$:

$$\begin{aligned}
\frac{\partial Q}{\partial \mathbf{W}} &= \frac{\partial}{\partial \mathbf{W}} \sum_{n=1}^{N} E_{z|x_n}\left[-\mathbf{x}_n^T \mathbf{\Psi}^{-1}\mathbf{x}_n + 2\mathbf{x}_n^T \mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z} - \mathbf{z}^T \mathbf{W}^T \mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z}\right] \\
&= \sum_{n=1}^{N} E_{z|x_n}\left[\frac{\partial}{\partial \mathbf{W}}\left(2\mathbf{x}_n^T \mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z} - \mathbf{z}^T \mathbf{W}^T \mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z}\right)\right] \\
&= \sum_{n=1}^{N} E_{z|x_n}\left[2\mathbf{\Psi}^{-1}\mathbf{x}_n\mathbf{z}^T - 2\mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z}\mathbf{z}^T\right] \\
&= 2\mathbf{\Psi}^{-1}\sum_{n=1}^{N}\left(\mathbf{x}_n E_{z|x_n}[\mathbf{z}^T] - \mathbf{W}E_{z|x_n}[\mathbf{z}\mathbf{z}^T]\right) = \mathbf{0} \qquad (11.56)
\end{aligned}$$

Solving for $\mathbf{W}$ we get

$$\mathbf{W} = \left(\sum_{n=1}^{N} \mathbf{x}_n E_{z|x_n}[\mathbf{z}^T]\right) \left(\sum_{n=1}^{N} E_{z|x_n}[\mathbf{z}\mathbf{z}^T]\right)^{-1} \tag{11.57}$$

where $E_{z|x_n}[\mathbf{z}]$ and $E_{z|x_n}[\mathbf{z}\mathbf{z}^T]$ can be found in Eq. (11.49):

$$\begin{cases} E_{z|x_n}[\mathbf{z}] &= \mathbf{m}_{z|x_n} = \mathbf{B}\mathbf{x}_n \\ E_{z|x_n}[\mathbf{z}\mathbf{z}^T] &= \mathbf{\Sigma}_{z|x_n} + \mathbf{m}_{z|x_n}\mathbf{m}_{z|x_n}^T = \mathbf{I} - \mathbf{B}\mathbf{W} + \mathbf{B}\mathbf{x}_n\mathbf{x}_n^T\mathbf{B}^T \end{cases} \tag{11.58}$$

The second equation is due to the fact that $\mathbf{\Sigma}_z = \mathrm{E}[\mathbf{z}\mathbf{z}^T] - \mathbf{m}_z\mathbf{m}_z^T$. Here $E_{z|x_n}[\mathbf{z}]$ can be considered as the estimation of the $\mathbf{z}$, while $E_{z|x_n}[\mathbf{z}\mathbf{z}^T]$ the uncertainty of the estimation.

– Find $\mathbf{\Psi}$:

$$\begin{aligned} \frac{\partial Q}{\partial \mathbf{\Psi}^{-1}} &= \frac{\partial}{\partial \mathbf{\Psi}^{-1}} \left[N \log|\mathbf{\Psi}| + \sum_{n=1}^{N} E_{z|x_n}\left[\mathbf{x}_n^T\mathbf{\Psi}^{-1}\mathbf{x}_n - 2\mathbf{x}_n^T\mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z} + \mathbf{z}^T\mathbf{W}^T\mathbf{\Psi}^{-1}\mathbf{W}\mathbf{z}\right]\right] \\ &= -N\mathbf{\Psi} + \sum_{n=1}^{N} E_{z|x_n}\left[\mathbf{x}_n\mathbf{x}_n^T - 2\mathbf{x}_n\mathbf{z}^T\mathbf{W}^T + \mathbf{W}(\mathbf{z}\mathbf{z}^T)\mathbf{W}^T\right] \\ &= -N\mathbf{\Psi} + \sum_{n=1}^{N} \mathbf{x}_n\mathbf{x}_n^T - 2\sum_{n=1}^{N} \mathbf{x}_n E_{z|x_n}[\mathbf{z}]^T\mathbf{W}^T + \mathbf{W}\sum_{n=1}^{N} E_{z|x}[\mathbf{z}\mathbf{z}^T]\mathbf{W}^T = 0 \end{aligned} \tag{11.59}$$

Solving for $\mathbf{\Psi}$ we get

$$\mathbf{\Psi} = \frac{1}{N}\,\mathrm{diag}\left(\sum_{n=1}^{N} \mathbf{x}_n\mathbf{x}_n^T - 2\left(\sum_{n=1}^{N} \mathbf{x}_n E_{z|x_n}[\mathbf{z}]^T\right)\mathbf{W}^T + \mathbf{W}\left(\sum_{n=1}^{N} E_{z|x_n}[\mathbf{z}\mathbf{z}^T]\right)\mathbf{W}^T\right) \tag{11.60}$$

where $\mathrm{diag}(\mathbf{A})$ is the operation that sets all off-diagonal elements of matrix $\mathbf{A}$ to zero, so that the resulting matrix is indeed diagonal as what $\mathbf{\Psi}$ should be. We further replace $\mathbf{W}$ in front of the last term above by that in Eq. (11.57) and get

$$\mathbf{\Psi} = \frac{1}{N}\,\mathrm{diag}\left(\sum_{n=1}^{N} \mathbf{x}_n\mathbf{x}_n^T - \sum_{n=1}^{N} \mathbf{x}_n E_{z|x_n}[\mathbf{z}]^T\mathbf{W}^T\right) \tag{11.61}$$

In summary, here are the steps of the EM algorithm:

1. Initialize parameters $\theta_{old} = \{\mathbf{W}_{old}, \mathbf{\Psi}_{old}\}$;
2. E-step:
   Find $E_{z|x}(\mathbf{z})$ and $E_{z|x}(\mathbf{z}\mathbf{z}^T)$ in Eq. (11.58) based on $\mathbf{m}_{z|x_n}$ and $\mathbf{\Sigma}_{z|x_n}$ in Eq. (11.49), which in turn is based on $\theta_{old} = \{\mathbf{W}_{old}, \mathbf{\Psi}_{old}\}$;
3. M-step:
   Find $\theta_{new} = \{\mathbf{W}_{new}, \mathbf{\Psi}_{new}\}$ in Eqs. (11.57) and (11.61), based on $E_{z|x}(\mathbf{z})$ and $E_{z|x}(\mathbf{z}\mathbf{z}^T)$;
4. Terminate if convergence criterion is satisfied, otherwise replace θ_{old} by θ_{new} and return to the E-step.

As the E-step and M-step of are interdependent on each other, they need to be carried out iteratively based on certain initial guess of the parameters in θ. In general, the EM algorithm is an iterative method for getting the maximum likelihood (ML) or maximum a posteriori (MAP) estimates of the parameters in a statistical model, based on some unobserved latent variables. The EM algorithm is widely used in machine learning and data science, such as in the method of probabilistic PCA considered in the following section, and the Gaussian mixture model for clustering analysis to be considered in Section 15.2.

11.4 Probabilistic PCA

The method of *probabilistic PCA (PPCA)* can be considered as a special case of factor analysis based on the model $\mathbf{x} = \mathbf{W}\mathbf{z} + \mathbf{e}$, when the random noise $\mathbf{e}$ is assumed to have an isotropic normal distributions with covariance matrix $\mathbf{\Psi} = \varepsilon\mathbf{I}$. More specially, if we further let $\varepsilon \to 0$ approach zero, the iteration of the EM algorithm for the PPCA becomes extremely simple, as we will see below.

Same as in Eqs. (11.37) and (11.39) for FA, here we also have the normal distribution of both $\mathbf{z}$ and $\mathbf{e}$:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I}), \qquad p(\mathbf{e}) = \mathcal{N}(\mathbf{0}, \varepsilon\mathbf{I}) \tag{11.62}$$

and the normal distribution of the observed data $\mathbf{x} = \mathbf{W}\mathbf{z} + \mathbf{e}$, same as in Eq. (11.43):

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{0}, \mathbf{W}\mathbf{W}^T + \varepsilon\mathbf{I}) \tag{11.63}$$

The goal here is to estimate ε and $\mathbf{W}$ as the model parameter, based on the observed data dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$.

Now Eqs. (11.49) through (11.51) are rewritten as:

$$\begin{cases} \mathbf{m}_{z|x_n} = E_{z|x_n}(\mathbf{z}) = \mathbf{B}\mathbf{x}_n \\ \mathbf{\Sigma}_{z|x_n} = Cov_{z|x_n}(\mathbf{z}) = \mathbf{I} - \mathbf{B}\mathbf{W} \end{cases} \tag{11.64}$$

$$\mathbf{B} = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \varepsilon\mathbf{I})^{-1} \tag{11.65}$$

$$(\mathbf{W}\mathbf{W}^T + \varepsilon\mathbf{I})^{-1} = \varepsilon^{-1}\mathbf{I} - \varepsilon^{-1}\mathbf{W}(\varepsilon\mathbf{I} + \mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T \tag{11.66}$$

The two steps of the EM algorithm become:

- E-step: Get the expectation of the log-likelihood by replacing $\mathbf{\Psi}$ in Eq. (11.54) by $\varepsilon\mathbf{I}$:

$$Q = -\frac{dN}{2}\log\varepsilon - \frac{1}{2\varepsilon}\sum_{n=1}^{N} E_{z|x_n}\left[\mathbf{x}_n^T\mathbf{x}_n - 2\mathbf{x}_n^T\mathbf{W}\mathbf{z} + \mathbf{z}^T\mathbf{W}^T\mathbf{W}\mathbf{z}\right] \tag{11.67}$$

- M-step: Find the optimal model parameters $\mathbf{W}$ and ε that maximize the expectation of the log-likelihood Q above, by setting the derivative of Q

with respect to the parameters to zero and solve the resulting equations. For $\mathbf{W}$, the result is the same as Eq. (11.57):

$$\mathbf{W}_{new} = \left(\sum_{n=1}^{N} \mathbf{x}_n E_{z|x_n}[\mathbf{z}^T]\right) \left(\sum_{n=1}^{N} E_{z|x_n}[\mathbf{z}\mathbf{z}^T]\right)^{-1} \tag{11.68}$$

where

$$\begin{cases} E_{z|x_n}[\mathbf{z}] &= \mathbf{m}_{z|x_n} = \mathbf{B}\mathbf{x}_n \\ E_{z|x_n}[\mathbf{z}\mathbf{z}^T] &= \mathbf{\Sigma}_{z|x_n} + \mathbf{m}_{z|x_n}\mathbf{m}_{z|x_n}^T \end{cases} \tag{11.69}$$

To find ε, we solve the equation

$$\frac{\partial Q}{\partial \varepsilon} = -\frac{dN}{2\varepsilon} + \frac{1}{2\varepsilon^2}\sum_{n=1}^{N} E_{z|x_n}\left[\mathbf{x}_n^T\mathbf{x}_n - 2\mathbf{x}_n^T\mathbf{W}\mathbf{z} + \mathbf{z}^T\mathbf{W}^T\mathbf{W}\mathbf{z}\right] = 0 \tag{11.70}$$

and get

$$\begin{aligned} \varepsilon &= \frac{1}{dN}\sum_{n=1}^{N}\left[\mathbf{x}_n^T\mathbf{x}_n - 2\mathbf{x}_n^T\mathbf{W}E_{z|x_n}(\mathbf{z}) + tr(\mathbf{W}^T E_{z|x_n}(\mathbf{z}\mathbf{z}^T)\mathbf{W})\right] \\ &= \frac{1}{dN}\sum_{n=1}^{N}\left[||\mathbf{x}_n||^2 - 2E_{z|x_n}(\mathbf{z}^T)\mathbf{W}^T\mathbf{x}_n + tr(E_{z|x_n}(\mathbf{z}\mathbf{z}^T)\mathbf{W}^T\mathbf{W})\right] \end{aligned} \tag{11.71}$$

where we have used the fact that $\mathbf{a}^T\mathbf{b} = tr(\mathbf{a}\mathbf{b}^T)$.

In summary, here are the steps in the EM algorithm for PPCA:

1. Initialize parameters $\theta_{old} = \{\mathbf{W}_{old}, \varepsilon_{old}\}$;
2. E-step: Based on θ_{old}, find $\mathbf{m}_{z|x_n}$ and $\mathbf{\Sigma}_{z|x_n}$ in Eq. (11.64), and $E_{z|x_n}(\mathbf{z})$ and $E_{z|x_n}(\mathbf{z}\mathbf{z}^T)$ in Eq. (11.69);
3. M-step: Find $\theta_{new} = \{\mathbf{W}_{new}, \varepsilon_{new}\}$ by evaluating Eqs. (11.68) and (11.71);
4. Terminate if convergence criterion is satisfied, otherwise replace θ_{old} by θ_{new} and return to step 2.

Specially, if we further assume $\varepsilon \to 0$ and the latent variables in $\mathbf{z}$ are deterministic, i.e., $E_{z|x}(\mathbf{z}) = \mathbf{z}$. At the limit where $\varepsilon = 0$, $\mathbf{x}$ is simply a linear combination of the latent variables in $\mathbf{z}$:

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \varepsilon\mathbf{I} \overset{\varepsilon\to 0}{\Longrightarrow} \mathbf{W}\mathbf{z} = [\mathbf{w}_1, \cdots, \mathbf{w}_{d'}]\begin{bmatrix} z_1 \\ \vdots \\ z_{d'} \end{bmatrix} = \sum_{i=1}^{d'} z_i\mathbf{w}_i \tag{11.72}$$

Comparing this expression of $\mathbf{x}$ with Eq. (10.45)

$$\mathbf{x} = \mathbf{V}'\mathbf{y} = \sum_{i=1}^{d'} y_i\mathbf{v}_i \tag{11.73}$$

we see that the column vectors in $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_{d'}]$ for the PPCA are similar to the column vectors of $\mathbf{V}$, the eigenvector matrix of $\mathbf{\Sigma}_x$, in the linear PCA,

as both of them can be viewed as the principal directions, an alternative set of basis vectors that also span the same d-dimensional feature space in which $\mathbf{x}$ resides. When $\mathbf{x}$ is expressed as a linear combination of these basis vectors, it can be approximated by only $d' < d$ of the d such basis vectors weighted by the d' principal components in $\mathbf{y} = [y_1, \cdots, y_{d'}]^T$ in linear PCA, or the d' factors in $\mathbf{z} = [z_1, \cdots, z_{d'}]^T$ in PPCA.

Here are the two EM steps for the PPCA:

- E-step: If $\varepsilon \to 0$, Eq. (11.64) can be written as:

$$\begin{aligned} \mathbf{m}_{z|x_n} &= E_{z|x_n}(\mathbf{z}) = \mathbf{z}_n = \mathbf{B}\mathbf{x}_n = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \varepsilon\mathbf{I})^{-1}\mathbf{x}_n \\ &\overset{\varepsilon\to 0}{\Longrightarrow} \mathbf{W}^T(\mathbf{W}\mathbf{W}^T)^{-1}\mathbf{x}_n \qquad (n = 1, \cdots, N) \end{aligned} \tag{11.74}$$

  All N such equations can be written in matrix form as

$$\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_n] = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T)^{-1}[\mathbf{x}_1, \cdots, \mathbf{x}_N] = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T)^{-1}\mathbf{X} \tag{11.75}$$

  As $\mathbf{W}$ is a $d \times d'$ matrix ($d' < d$), the rank of the $d \times d$ matrix $\mathbf{W}\mathbf{W}^T$ is at most d', i.e., the inverse of $\mathbf{W}\mathbf{W}^T$ does not exist and we cannot find $\mathbf{z}_n = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T)^{-1}\mathbf{x}_n$. In fact the transformation matrix $\mathbf{W}^T(\mathbf{W}\mathbf{W}^T)^{-1}$ is the *right pseudo inverse* (Section A.6.1) of the $d \times d'$ matrix $\mathbf{W}$, which exists only if $d' \geq d$.

  However, we note that the model $\mathbf{x} = \mathbf{W}\mathbf{z}$ is an over-constrained linear system of d equations but only $d' < d$ variables in $\mathbf{z} = [z_1, \cdots, z_{d'}]^T$, and the least-squares solution that minimizes the squared error $||\mathbf{x} - \mathbf{W}\mathbf{z}||^2$ can be found by the *left pseudo inverse* (Section A.6.1) of $\mathbf{W}$:

$$\mathbf{z}_n = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{x}_n = \mathbf{W}^-\mathbf{x}_n \qquad (n = 1, \cdots, N) \tag{11.76}$$

  and we can get the latent variables in $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_N]$ given all N data points in $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$:

$$\mathbf{Z} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{X} = \mathbf{W}^-\mathbf{X} \tag{11.77}$$

  which is based on the old value of $\mathbf{W}$ previously estimated.

- M-step: Now Eq. (11.68) can be written as:

$$\begin{aligned} \mathbf{W} &= \left(\sum_{n=1}^{N} \mathbf{x}_n E_{z|x_n}(\mathbf{z}^T)\right) \left(\sum_{n=1}^{N} E_{z|x_n}(\mathbf{z}\mathbf{z}^T)\right)^{-1} \\ &= [\mathbf{x}_1, \cdots, \mathbf{x}_N] \begin{bmatrix} \mathbf{z}_1^T \\ \vdots \\ \mathbf{z}_N^T \end{bmatrix} \left([\mathbf{z}_1, \cdots, \mathbf{z}_N] \begin{bmatrix} \mathbf{z}_1^T \\ \vdots \\ \mathbf{z}_N^T \end{bmatrix} \right)^{-1} \\ &= \mathbf{X}\mathbf{Z}^T(\mathbf{Z}\mathbf{Z}^T)^{-1} = \mathbf{X}\mathbf{Z}^- \end{aligned} \tag{11.78}$$

  by which $\mathbf{W}$ is updated based on $\mathbf{Z}$ found previously in the E-step. We note that $\mathbf{Z}^T(\mathbf{Z}\mathbf{Z}^T)^{-1}$ is the right pseudo inverse of the $d \times N$ matrix $\mathbf{Z}$, which exists if $d \leq N$.

Again, as the E-step and M-steps are interdependent on each other, they need to be carried out iteratively:

$$\begin{aligned} \text{E-step:} \quad & \mathbf{Z} = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{X} = \mathbf{W}^-\mathbf{X} \\ \text{M-Step:} \quad & \mathbf{W} = \mathbf{X}\mathbf{Z}^T(\mathbf{Z}\mathbf{Z}^T)^{-1} = \mathbf{X}\mathbf{Z}^- \end{aligned} \tag{11.79}$$

We note that given the dataset $\mathbf{X}$, the FA model $\mathbf{x} = \mathbf{W}\mathbf{z}$ can be expressed as an over-determined matrix equation

$$\mathbf{X}_{d\times N} = [\mathbf{x}_1, \cdots, \mathbf{x}_N] = \mathbf{W}_{d\times d'}[\mathbf{z}_1, \cdots, \mathbf{z}_N] = \mathbf{W}_{d\times d'}\mathbf{Z}_{d'\times N} \tag{11.80}$$

or, taking transpose of the equation, we get

$$\mathbf{X}^T_{N\times d} = \mathbf{Z}^T_{N\times d'}\mathbf{W}^T_{d'\times d} \tag{11.81}$$

Given $\mathbf{X}$, this equation can be interpreted in two ways:

- If $\mathbf{W}$ is available, this is a system of d equations but $d' < d$ unknowns in each of the N columns of $\mathbf{Z}$, which is solved as in the E-step above;
- If $\mathbf{Z}$ is available, this is a system of N equations but $d' < N$ unknowns in each of the d rows of $\mathbf{W}$, which is solved as in the M-step above.

In either step, the pseudo inverse method can be used to minimize the squared error $||\mathbf{X} - \mathbf{W}\mathbf{Z}||^2$.

When compared to the regular PCA method that finds all d eigenvalues and the corresponding eigenvectors all at once by solving the eigenequation $\mathbf{\Sigma}_x\mathbf{w} = \lambda\mathbf{w}$, the PPCA method only finds the d' column vectors of matrix $\mathbf{W}$ as the basis vectors, as the basis vectors that span a subspace with much reduced dimensionality but containing the most essential information in the data. However, unlike the PCA, the basis vectors in PPCA are not necessarily orthogonal to each other.

The implementation of the PPCA is extremely simple, as shown in the Metlab code below. Based on some initialized $\mathbf{W}$, the EM iteration is composed of the E-step and M-step:

```
while er>tol
    Z=inv(W'*W)*W'*X;       % find Z given W in E-step
    Wnew=X*Z'*inv(Z*Z');    % find W given Z in M-step
    er=norm(Wnew-W);        % the LS error
    W=Wnew;
end
```

Figs. 11.5 and 11.6 show the first few iterations of the PPCA based on the EM method presented above, for a set of data points in $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ in 2-D and 3-D. The straight lines in the plots represents the column vectors of $\mathbf{W}$, which is randomly initialized, but quickly approach the directions corresponding to the principal components obtained by the PCA method, along which the data points are most widely spread.

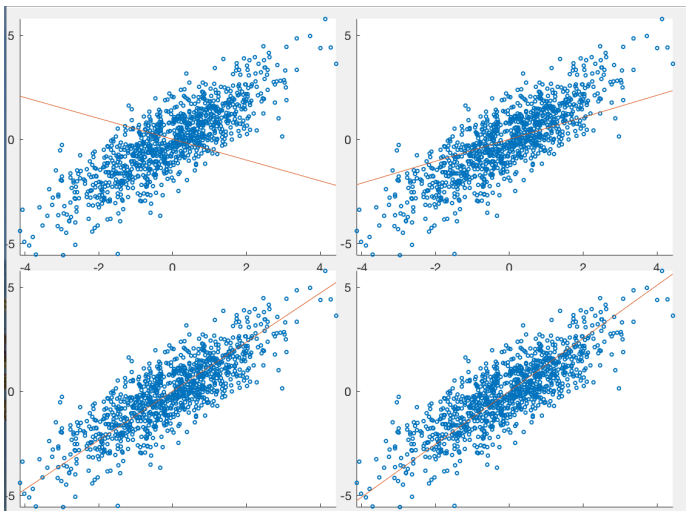


Figure 11.5 Column vectors in $\mathbf{W}$ iteratively approach the principal directions of dataset (2-D)

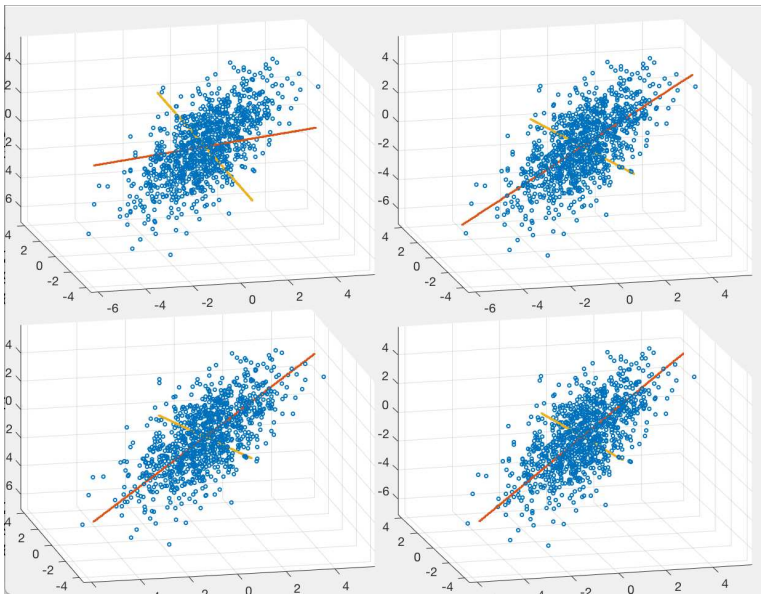


Figure 11.6 Column vectors in $\mathbf{W}$ iteratively approach the principal directions of dataset (3-D)

Fig. 11.7 shows the dataset of 10 hand-written numbers from 0 to 9, containing 2240 data points in a 256-D space, but now linearly mapped to a 3-D space spanned by $d' = 3$ factors found by the PPCA method.

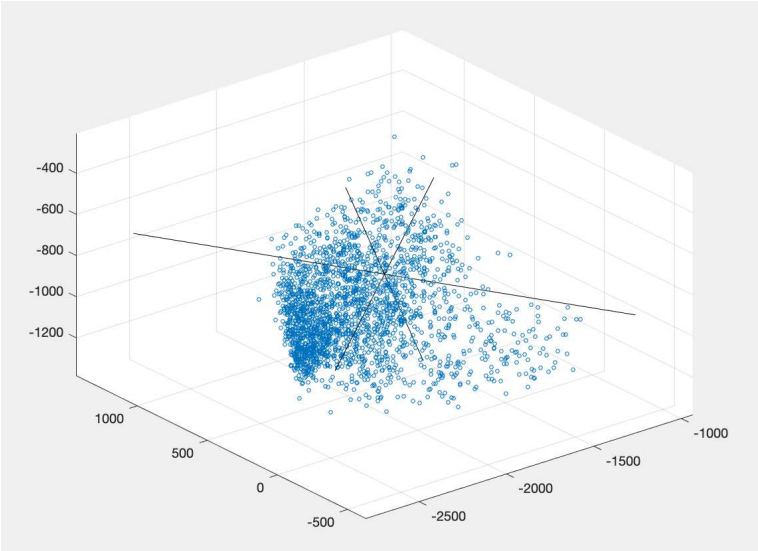


Figure 11.7 First three components of the handwritten digits dataset by the PPCA

11.5 Classical Multidimensional Scaling

The goal of *multidimensional scaling (MDS)* is to reduce the dimensionality of the dataset representing a set of N objects of interest, each described by a set of d features and represented as a vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ in a d-dimensional space, while the pairwise similarity relationship between any two of these objects are preserved. MDS can be used to visualize datasets in a high dimensional space. Specially, here we consider the *classical MDS (cMDS)*, also known as *principal coordinates analysis (PCoA)*, as one of the methods in MDS.

Specifically, we represent the pairwise similarity of a set of N objects $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ by the $N \times N$ similarity matrix $\mathbf{D}_x = [\ d_{ij}^x\ ]$, of which the ij-th component d_{ij}^x is certain measurement of the similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$. The goal is to map the dataset $\mathbf{X}$ in the d-dimensional space into $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$ in a lower d'-dimensional space, in which the similarity matrix $\mathbf{D}_x$ is optimally approximated by $\mathbf{D}_y = [\ d_{ij}^y]$. When $d' \leq 3$, these data points in $\mathbf{Y}$ can be visualized. MDS can be considered as a method for dimension reduction. In some cases only the pairwise similarities d_{ij}^x $(i, j = 1, \cdots, N)$ are given, while the N objects in $\mathbf{X}$ are not explicitly given, and the dimensionality d may be unknown.

We assume the given pairwise similarity is the Euclidean distance $d_{ij} = ||\mathbf{x}_i - \mathbf{x}_j||_2$ between $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$d_{ij}^2 = ||\mathbf{x}_i - \mathbf{x}_j||^2 = (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_i - 2\mathbf{x}_i^T\mathbf{x}_j + \mathbf{x}_j^T\mathbf{x}_j \qquad (11.82)$$

and get

$$\mathbf{D}_x^2 = \begin{bmatrix} & d_{ij}^2 & \end{bmatrix}_{N\times N} = \begin{bmatrix} & \mathbf{x}_i^T\mathbf{x}_i - 2\mathbf{x}_i^T\mathbf{x}_j + \mathbf{x}_j^T\mathbf{x}_j & \end{bmatrix}_{N\times N} = \mathbf{X}_r - 2\mathbf{X}^T\mathbf{X} + \mathbf{X}_c \tag{11.83}$$

where

$$\mathbf{X}_r = \begin{bmatrix} ||\mathbf{x}_1||^2 & \cdots & ||\mathbf{x}_1||^2 \\ \vdots & \ddots & \vdots \\ ||\mathbf{x}_N||^2 & \cdots & ||\mathbf{x}_N||^2 \end{bmatrix} = \mathbf{X}_c^T$$

$$\mathbf{X}^T\mathbf{X} = \begin{bmatrix} \mathbf{x_1}^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} [\mathbf{x}_1, \cdots, \mathbf{x}_N] = \begin{bmatrix} \mathbf{x}_1^T\mathbf{x}_1 & \cdots & \mathbf{x}_1^T\mathbf{x}_N \\ \vdots & \ddots & \vdots \\ \mathbf{x}_N^T\mathbf{x}_1 & \cdots & \mathbf{x}_N^T\mathbf{x}_N \end{bmatrix} \tag{11.84}$$

Our goal is to find $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$ in a lower dimensional space so that its pairwise similarity matrix $\mathbf{D}_y$ matches $\mathbf{D}_x$ optimally, in the sense that the following objective function is minimized:

$$J(\mathbf{Y}) = ||\mathbf{D}_y^2 - \mathbf{D}_x^2|| \tag{11.85}$$

We note that such a solution $\mathbf{Y}$ is not unique, as any translated version of $\mathbf{Y}$ has the same pairwise similarity matrix and is also a solution. We therefore seek to find a unique solution $\mathbf{Y}$ in which all data points are centralized, i.e, the mean of all points in $\mathbf{Y}$ is zero:

$$\bar{\mathbf{y}} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{y}_n = \mathbf{0} \tag{11.86}$$

To do so, we introduce an $N \times N$ symmetric centering matrix $\mathbf{C} = \mathbf{I} - \mathbf{1}/N = \mathbf{C}^T$ (Section A.8.1), where$\mathbf{1} = \mathbf{e}\mathbf{e}^T$ is an $N \times N$ matrix with all components equal 1. Premultiplying $\mathbf{C}$ to a column vector $\mathbf{a} = [a_1, \cdots, a_N]^T$ removes the mean $\bar{a} = \sum_{i=1}^{N} a_i/N$ of $\mathbf{a}$ from each of its component:

$$\mathbf{Ca} = \mathbf{Ia} - \frac{1}{N}\mathbf{1a} = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} - \frac{1}{N} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} a_1 - \bar{a} \\ \vdots \\ a_N - \bar{a} \end{bmatrix} \tag{11.87}$$

Taking transpose of the equation above, we get

$$(\mathbf{Ca})^T = \mathbf{a}^T\mathbf{C} = [a_1 - \bar{a}, \cdots, a_N - \bar{a}] \tag{11.88}$$

i.e., postmultiplying $\mathbf{C}$ to a row vector $\mathbf{a}^T = [a_1, \cdots, a_N]$ removes the mean $\bar{a}$ from each of its component.

We can now postmultiply $\mathbf{C}$ to $\mathbf{X}$ to remove the mean of each row of $\mathbf{X}$:

$$\mathbf{XC} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]\mathbf{C} = [\bar{\mathbf{x}}_1, \cdots, \bar{\mathbf{x}}_N] = \bar{\mathbf{X}} \tag{11.89}$$

i.e., the nth column becomes:

$$\bar{\mathbf{x}}_n = \mathbf{x}_n - \bar{\mathbf{x}}, \quad \text{where} \quad \bar{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \tag{11.90}$$

and premultiply $\mathbf{C}$ to $\mathbf{X}^T$ to remove the mean of each column of $\mathbf{X}^T$:

$$\mathbf{C}\mathbf{X}^T = \bar{\mathbf{X}}^T \tag{11.91}$$

We further apply double centering to $\mathbf{D}_x^2$ by pre and post-multiplying $\mathbf{C}$ and get:

$$\begin{aligned}\mathbf{C}\mathbf{D}_x^2\mathbf{C} &= \mathbf{C}\left(\mathbf{X}_r - 2\mathbf{X}^T\mathbf{X} + \mathbf{X}_c\right)\mathbf{C} = \mathbf{C}\mathbf{X}_r\mathbf{C} - 2\mathbf{C}\left(\mathbf{X}^T\mathbf{X}\right)\mathbf{C} + \mathbf{C}\mathbf{X}_c\mathbf{C} \\ &= -2\mathbf{C}\left(\mathbf{X}^T\mathbf{X}\right)\mathbf{C} = -2\left(\mathbf{X}\mathbf{C}\right)^T\left(\mathbf{X}\mathbf{C}\right) = -2\bar{\mathbf{X}}^T\bar{\mathbf{X}}\end{aligned}$$

Note that the first term $\mathbf{C}\mathbf{X}_r\mathbf{C} = \mathbf{C}(\mathbf{X}_r\mathbf{C}) = \mathbf{C}\mathbf{0} = \mathbf{0}$, as all components of each row of $\mathbf{X}_r$ are the same as their mean, and removing the mean results in a zero row vector, $\mathbf{X}_r\mathbf{C} = \mathbf{0}$; and the last term $\mathbf{C}\mathbf{X}_c\mathbf{C} = (\mathbf{C}\mathbf{X}_c)\mathbf{C} = \mathbf{0}\mathbf{C} = \mathbf{0}$, as all components of each column of $\mathbf{X}_c$ are the same as their mean, and removing the mean results in a zero column vector.

Now the similarity matrix $\mathbf{D}_x^2$ of all centralized data points in $\bar{\mathbf{X}}$ with zero mean can be represented by the *Gram matrix* of $\bar{\mathbf{X}}$:

$$\mathbf{G}_{\bar{x}} = \bar{\mathbf{X}}^T\bar{\mathbf{X}} = -\frac{1}{2}\mathbf{C}\mathbf{D}_x^2\mathbf{C} \tag{11.92}$$

We further assume all data points in $\mathbf{Y}$ are also centered with zero mean and their corresponding similarity matrix is also represented by $\mathbf{G}_{\bar{y}}$, then the objective function can be expressed as:

$$J(\mathbf{Y}) = ||\mathbf{G}_{\bar{y}} - \mathbf{G}_{\bar{x}}|| = ||\bar{\mathbf{Y}}^T\bar{\mathbf{Y}} - \mathbf{G}_{\bar{x}}|| \tag{11.93}$$

To find $\bar{\mathbf{Y}}$ that minimizes $J(\mathbf{Y})$, we first consider the equation $\bar{\mathbf{Y}}^T\bar{\mathbf{Y}} - \mathbf{G}_{\bar{x}} = \mathbf{0}$ so that $J(\mathbf{Y}) = 0$. In other words, we desire to express $\mathbf{G}_{\bar{x}}$ as the inner product of some matrix with itself. To do so, we carry out eigenvalue decomposition of $\mathbf{G}_x$ to find its eigenvalue matrix $\mathbf{\Lambda}$ and the orthogonal eigenvector matrix $\mathbf{V}$ satisfying $\mathbf{G}_x\mathbf{V} = \mathbf{V}\mathbf{\Lambda}$, i.e.,

$$\begin{aligned}\mathbf{G}_{\bar{x}} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T &= [\mathbf{v}_1, \cdots, \mathbf{v}_N]\begin{bmatrix}\lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_N\end{bmatrix}\begin{bmatrix}\mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_N^T\end{bmatrix} \\ = \left(\mathbf{V}\mathbf{\Lambda}^{1/2}\right)\left(\mathbf{\Lambda}^{1/2}\mathbf{V}^T\right) &= \left(\mathbf{\Lambda}^{1/2}\mathbf{V}^T\right)^T\left(\mathbf{\Lambda}^{1/2}\mathbf{V}^T\right)\end{aligned}$$

Note that as $\mathbf{G}_{\bar{x}}$ is symmetric, the eigenvalues are real and the eigenvectors are orthogonal, i.e., $\mathbf{V}^T = \mathbf{V}^1$ is is orthogonal matrix. Now we can find the desired $\mathbf{Y}$ as:

$$\mathbf{Y} = \mathbf{\Lambda}^{1/2}\mathbf{V}^T = \begin{bmatrix}\sqrt{\lambda_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sqrt{\lambda_N}\end{bmatrix}\begin{bmatrix}\mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_N^T\end{bmatrix} = \begin{bmatrix}\sqrt{\lambda_1}\mathbf{v}_1^T \\ \vdots \\ \sqrt{\lambda_N}\mathbf{v}_N^T\end{bmatrix} \tag{11.94}$$

Each column $\mathbf{y}_i$ in $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$ is an N-D vector. If all N dimensions of these $\mathbf{y}_n$ are used, $\mathbf{Y}^T\mathbf{Y} = \mathbf{G}_x$ and $J(\mathbf{Y} = 0$. To reduce the dimensionality to $d' < d$, we construct $\mathbf{Y}'$ by the first d' rows of $\mathbf{Y}$ corresponding to the d' greatest eigenvalues of $\lambda_1 \geq \cdots \geq \lambda_{d'} \geq \cdots \geq \lambda_N$ and their corresponding eigenvectors $\mathbf{v}_1, \cdots, \mathbf{v}_{d'}$. The error is:

$$J(\mathbf{Y}') = ||\mathbf{Y}'^T\mathbf{Y}' - \mathbf{Y}^T\mathbf{Y}|| = ||\sum_{i=1}^{d'} \lambda_i \mathbf{v}_i \mathbf{v}^T - \sum_{i=1}^{N} \lambda_i \mathbf{v}_i \mathbf{v}^T|| = ||\sum_{i=d'}^{N} \lambda_i \mathbf{v}_i \mathbf{v}^T|| \tag{11.95}$$

In summary, here is the PCoA algorithm:

- Given the pairwise similarity, an $N \times N$ matrix $\mathbf{D}_x = [\ d_{ij}\ ]$, construct the squared proximity matrix $\mathbf{D}_x^2 = [\ d_{ij}^2\ ]$;
- Apply double centering to get $\mathbf{G}_x = -\mathbf{C}\mathbf{D}_x^2\mathbf{C}/2$;
- Find the d' greatest eigenvalues $\lambda_1, \cdots, \lambda_{d'}$ of $\mathbf{G}_x$, and the corresponding eigenvectors $\mathbf{v}_1, \cdots, \mathbf{v}_{d'}$;
- Get the map

$$\mathbf{Y}_{d' \times N} = \mathbf{\Lambda}_{d'}^{1/2}\mathbf{V}_{d'} = \begin{bmatrix} \sqrt{\lambda_1}\mathbf{v}_1^T \\ \vdots \\ \sqrt{\lambda_{d'}}\mathbf{v}_{d'}^T \end{bmatrix} \tag{11.96}$$

The Metlab functions for the PCA and PCoA are listed below:

```
function Y=PCA(X)
    [V D]=eig(cov(X'));        % solve eigenequation of covariance
    [d idx]=sort(diag(D),'descend');
                               % sort eigenvalues in descend order
    V=V(:,idx);                % reorder corresponding eigenvectors
    Y=V'*X;                    % carry out KLT
end

function Y=PCoA(X)
    N=size(X,2);
    D=zeros(N);                % pairwise similarity matrix
    C=eye(N)-ones(N)/N;        % centering matrix
    for i=1:N
        for j=1:N
            D(i,j)=(norm(X(:,i)-X(:,j)))^2;
        end
    end
    B=-C*D*C/2;
    [V D]=eig(B);              % solve eigenequation for matrix B
    [d idx]=sort(diag(D),'descend');
                               % sort eigenvalues in descend order
    V=V(:,idx);                % reorder eigenvectors
```

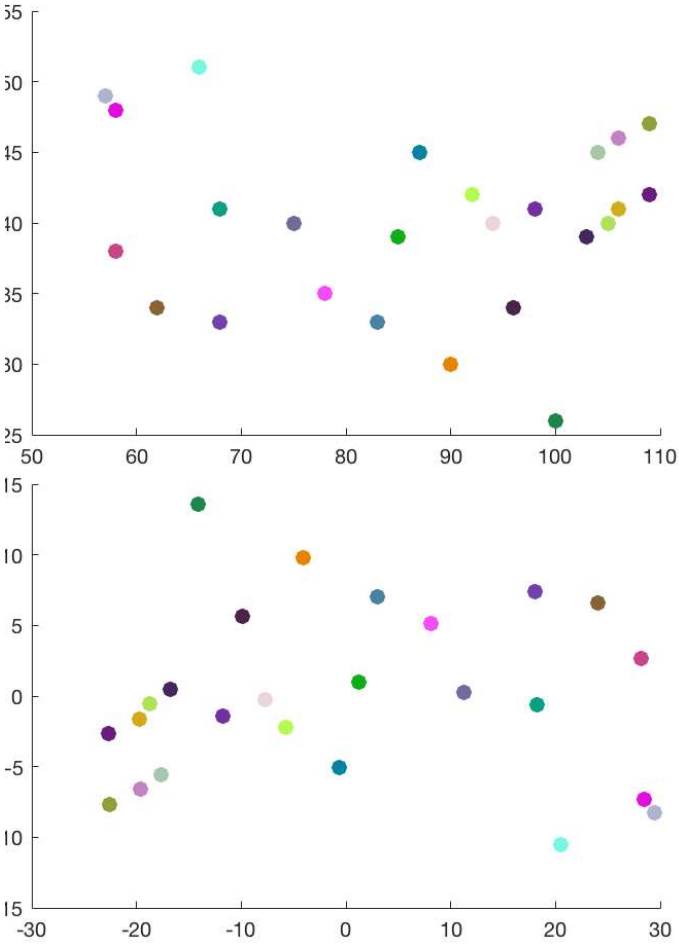


Figure 11.8 Reconstruction of the locations of north America cities by the MDS

```
    Y=sqrt(D)*V';                % carry out PCoA
end
```

In Fig. 11.8, the top panel shows the locations of some major cities in North America, and the bottom panel shows the reconstructed city locations by the MDS method based on the pairwise distance of these cities. Note that the reconstructed map is a rotated and centralized version of the original map.

The methods of PCA and PCoA are compared when applied to the 4-D iris dataset and the 256-D handwritten digits dataset, with their corresponding visualizations in both 2-D and 3-D shown in Figs. 11.9 and 11.10.

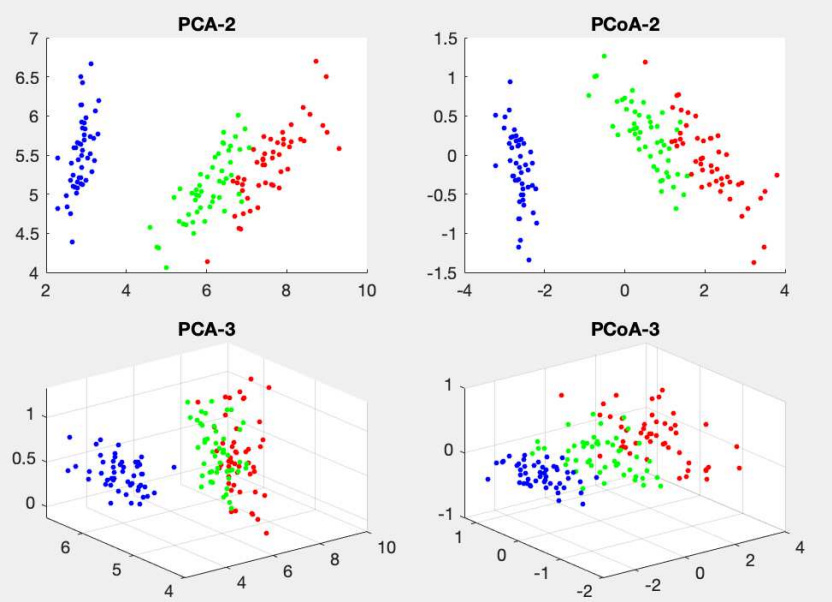


Figure 11.9 Visualization of the iris dataset based on both PCA and PCoA

11.6 t-Distributed Stochastic Neighbor Embedding

The method of PCA considered previously is often used to reduce the dimensionality of a given dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ by an orthogonal mapping from the high dimensional space to a low dimensional space. However, sometimes this method based on the linear KLT transform may not be able to properly interpret complex nonlinear relationship between the variables that spanned the feature space, as it may not be able to maintain the relative pairwise distances between the data points in the resulting lower dimensional space. For example, in the low dimensional space after the transform, those dissimilar data points far apart from each other in a large scale in the original high dimensional space may be well represented in the low dimensional space by the major principal components along the axes with large variances, but the differences between those points dissimilar with each other in some smaller scales may not be properly represented, or even completely neglected, if their dissimilarity is mostly represented in the dimensions with small variances. In other words, the PCA method tends to emphasize large scale global structures of the dataset, while ignoring the small scale local structures, which may be equally important or even of greater interest, and therefore also need to be properly represented.

Also, when the dimensionality is much reduced from a high-D space to a low-D space, the crowding problem occurs. In a high-D space, a data point may have a large number of similar neighbors of roughly equal distance; however, in a low-D space, the number of equal-distance neighbors is significantly reduced (e.g., 8

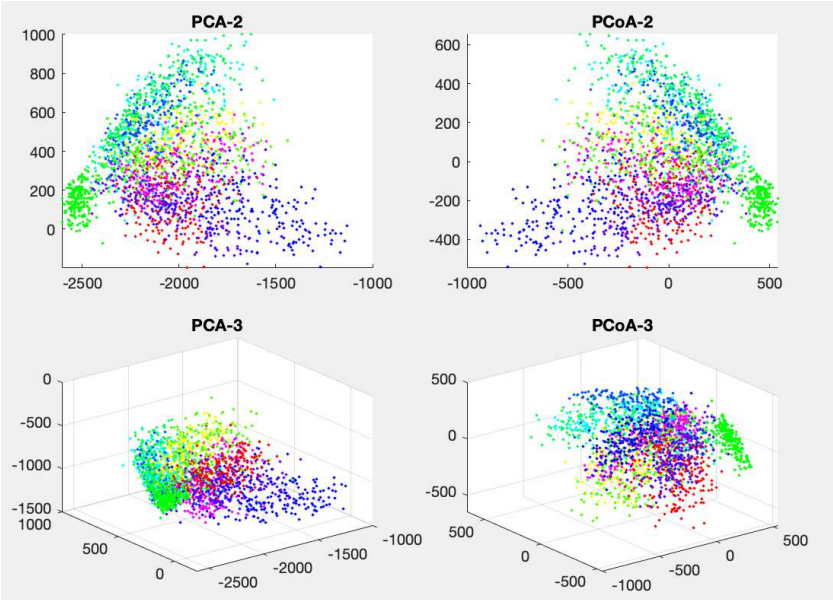


Figure 11.10 Visualization of the handwritten digits dataset based on both PCA and PCoA

immediate neighbors in a 3-D space, 4 in a 2-D space, 2 in a 1-D space). Thus the space available for the neighbors of a point is reduced while the dimensionality is reduced. As the result, when a large number of neighbors in the high-D are mapped to a low-D space, they are forced to spread out, some may get closer to those that are originally farther away in the high-D space, thereby reducing the gaps between potential clusters of similar points.

In contrast, the t-Distributed Stochastic Neighbor Embedding (tSNE) method as a nonlinear method that is based on probability distributions of the data points being neighbors, and it attempts to preserve the structure at all scales, but emphasizing more on the small scale structures, by mapping nearby points in high-D space to nearby points in low-D space.

In the high-D space, the similarity between any data point $\mathbf{x}_i$ and each of other points $\mathbf{x}_j$ in the dataset $\mathbf{X}$ is defined based on the Gaussian kernel as a conditional probability:

$$p_{j/i} = \frac{\exp(-||\mathbf{x}_i - \mathbf{x}_j||^2/2\sigma_i^2)}{\sum_{k=1}^{N}\sum_{l=1}^{N}\exp(-||\mathbf{x}_k - \mathbf{x}_l||^2/2\sigma_i^2)} \tag{11.97}$$

where σ_i^2 is the variance of the data points in the local neighborhood of $\mathbf{x}_i$.

We see that if any $\mathbf{x}_j$ is close to $\mathbf{x}_i$, $||\mathbf{x}_i - \mathbf{x}_j||$ is small and $p_{j/i}$ is large, indicating $\mathbf{x}_j$ is similar to $\mathbf{x}_i$; but if $\mathbf{x}_j$ is far away from $\mathbf{x}_j$, $||\mathbf{x}_i - \mathbf{x}_j||$ is large and p_{ij} is small, indicating $\mathbf{x}_j$ is dissimilar to $\mathbf{x}_i$. Also, as $\sum_i \sum_j p_{ij} = 1$, i.e., p_{ij} is normalized and can therefore be considered as a probability that any two points

$\mathbf{x}_j$ and $\mathbf{x}_i$ are similar to each other and are therefore neighbors in the feature space.

As a similarity measurement between any two points $\mathbf{x}_i$ and $\mathbf{x}_j$, the conditional probability $p_{j/i}$ defined above needs to be modified for two reasons.

First, the Gaussian kernel p_{ij} should be independent of the local density of the data points, which may vary drastically in different local regions of the feature space, so that similarities of different scales can be equally well represented. To achieve this goal, we adjust the variance σ_i^2 in the Gaussian kernel $p_{j/i}$ for each data point $\mathbf{x}_i$ in such a way that the entropy $H(\mathbf{x}_i) = -\sum_j p_{j/i} \log_2 p_{j/i}$ for all data points are the same as some prespecified value. As shown in Section B.2.1, the entropy of the Gaussian distribution $\mathcal{N}(x, \mu, \sigma)$ is $\log(2\pi e\sigma^2)/2$, a monotonic function of the variance σ^2. The similarity measurement based on $p_{j/i}$ of the same entropy for all data points is essentially the same as the Mahalanobis distance (Eq. (9.12) and the Bhattacharyya distance (Eq. (9.16)), in the sense that these similarity measurements are all normalized by the variance σ^2, and they are therefore independent with respect to the local data density and scale.

Second, as $p_{j/i} \neq p_{i/j}$ as a conditional probability defined above is asymmetrical, it needs to be modified so that it becomes symmetric as a similarity measurement should be. We therefore redefine the probability measurement as the average of two conditional probabilities:

$$p_{ij} = \frac{p_{i/j} + p_{j/i}}{2} \tag{11.98}$$

Once the dataset $\mathbf{X}$ in the high dimensional space is mapped to $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$ in the low dimensional space, we also need to measure the similarity between any two data points. To do so, we further define such a similarity based on Student's t-distribution of degree 1 (Eq. (B.140)):

$$q_{ij} = \frac{(1 + ||\mathbf{y}_i - \mathbf{y}_j||^2)^{-1}}{\sum_{k=1}^N \sum_{l=1}^N (1 + ||\mathbf{y}_k - \mathbf{y}_l||^2)^{-1}} = \frac{(1 + d_{ij}^2)^{-1}}{\sum_{k=1}^N \sum_{l=1}^N (1 + d_{ij}^2)^{-1}} = \frac{(1 + d_{ij}^2)^{-1}}{D} \tag{11.99}$$

where D denotes the denominator:

$$D = \sum_{k=1}^N \sum_{l=1}^N (1 + d_{kl}^2)^{-1} \tag{11.100}$$

and $d_{kl} = ||\mathbf{y}_k - \mathbf{y}_l|| = \left[(\mathbf{y}_k - \mathbf{y}_l)^T (\mathbf{y}_k - \mathbf{y}_l)\right]^{1/2}$ is the Euclidean distance between any two data points $\mathbf{y}_k$ and $\mathbf{y}_l$. Again as q_{ij} is normalized, it can be interpreted as a probability measurement of the similarity between any two points in the low dimensional space.

The t-distribution with one degree of freedom is also a bell shaped function, similar to the Gaussian distribution, but it has a lower peak in the center but higher tails on the sides, as shown in Fig. 11.11, which also illustrates why the similarity is measure based on the Gaussian distribution in the high-D space but

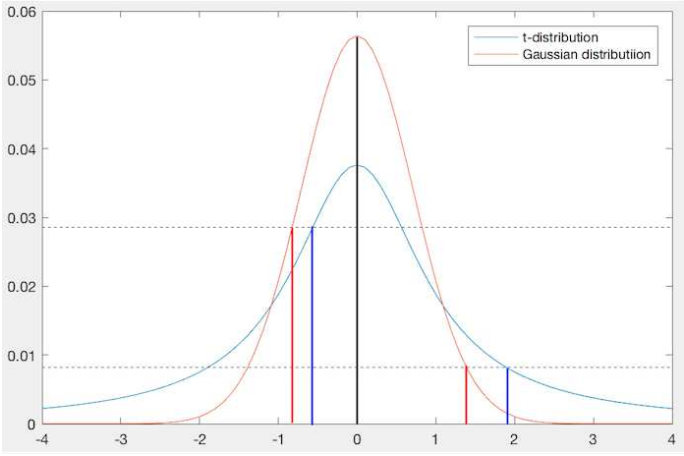


Figure 11.11 The t-distribution compared to the Gaussian distribution

the t-distribution in the low-D space. Specifically, consider any point x_i and two of its neighbors x_j and x'_j in the high-D space also illustrated in the figure:

- Point $x_j = -0.9$ (red on the left) similar to $x_i = 0$ as measured by the Gaussian in the high-D space is mapped to $y_j = -0.6$ in the low-D space measured by the t-distribution, even closer to $y_i = x_i$.
- Point $x'_j = 1.3$ (red on the right) not very similar to $x_i = 0$ as measured by the Gaussian in the high-D space is mapped to $y'_j = 1.9$ in the low-D space measured by the t-distribution, even farther away from $y_i = x_i$.

In other words, when all data points x_n, $(n = 1, \cdots, N)$ in the high-D space are nonlinearly mapped to y_n, $(n = 1, \cdots, N)$ in the low-D space in such a way that their similarities measured by the Gaussian before the mapping match those measured by the t-distribution after the mapping, similar points become even closer while dissimilar points become farther apart from each other. We note that whether two points are considered to be similar or not depends on the variance of the Gaussian, a hyperparameter that needs to be properly determined based on the specific dataset in question.

The goal of the tSNE method is to find the optimal mapping from all points in dataset $\mathbf{X}$ in the high-D space to $\mathbf{Y}$ in the low-D space that minimizes the discrepancy between the two sets of similarity measurements p_{ij} before the mapping and q_{ij} after the mapping, which can be quantitatively measured by the Kullback-Leibler divergence (Section B.2.2):

$$J = D_{KL}(P||Q) = \sum_{i=1}^{N}\sum_{j=1}^{N} p_{ij} \log \frac{p_{ij}}{q_{ij}} = \sum_{i=1}^{N}\sum_{j=1}^{N} (p_{ij} \log p_{ij} - p_{ij} \log q_{ij}) \tag{11.101}$$

The KL-divergence is not symmetric as $D_{KL}(P||Q) \neq D_{KL}(Q||P)$, i.e., it can

not be viewed as a distance between the two sets of probabilities. However, this asymmetry is actually a desired feature. Consider the following two cases:

- If p_{ij} is large but q_{ij} is small, then D_{KL} is large, and such a mapping is heavily penalized. As the result, similar points that are close to each other in the high-D space tend to be mapped to points also close to each other in the low-D space.
- If p_{ij} is small but q_{ij} is large, then D_{KL} is small, and such a mapping is not heavily penalized. As the result, dissimilar points that are far apart in the high-D space are allowed to be mapped to points closer to each other in the low-D space.

We therefore see that minimizing the KL-divergence has the effect of preserving the local structure of the dataset, but not necessarily the global structure.

The optimal points in $\mathbf{Y}$ in the low-D space that minimize the objective function $J = D_{KL}(P||Q)$ can be found by the gradient descent method. The ith component $\partial J/\partial \mathbf{y}_i$ of the gradient can be found by the chain rule, based on $d_{ij} = d_{ji}$ treated as an intermediate variable:

$$\frac{\partial J}{\partial \mathbf{y}_i} = \sum_{j=1}^{N} \left(\frac{\partial J}{\partial d_{ij}} \frac{\partial d_{ij}}{\partial \mathbf{y}_i} + \frac{\partial J}{\partial d_{ij}} \frac{\partial d_{ij}}{\partial \mathbf{y}_i} \right) = 2 \sum_{j=1}^{N} \frac{\partial J}{\partial d_{ij}} \frac{\partial d_{ij}}{\partial \mathbf{y}_i} \tag{11.102}$$

Note that $J = D_{KL}(P||Q)$ is a function of both p_{ij} as a constant, and $q_{ij} = (1 + d_{ij}^{-1})/D$, which in turn is a function of d_{ij}. The first derivative is:

$$\begin{aligned} \frac{\partial J}{\partial d_{ij}} &= \frac{\partial}{\partial d_{ij}} \left[\sum_{k=1}^{N} \sum_{l=1}^{N} (p_{kl} \log p_{kl} - p_{kl} \log q_{kl}) \right] \\ &= -\frac{\partial}{\partial d_{ij}} \sum_{k=1}^{N} \sum_{l=1}^{N} p_{kl} \log q_{kl} = -\frac{\partial}{\partial d_{ij}} \sum_{k=1}^{N} \sum_{l=1}^{N} p_{kl} (\log(q_{kl} D) - \log D) \\ &= -\sum_{k=1}^{N} \sum_{l=1}^{N} p_{kl} \frac{\partial}{\partial d_{ij}} \log(q_{kl} D) + \sum_{k=1}^{N} \sum_{l=1}^{N} p_{kl} \frac{1}{D} \frac{\partial D}{\partial d_{ij}} \end{aligned} \tag{11.103}$$

To proceed, we consider the two terms separately. There is only one non-zero term in the summation of the first term:

$$p_{ij} \frac{1}{q_{ij} D} \frac{\partial}{\partial d_{ij}} (1 + d_{ij}^2)^{-1} = \frac{p_{ij}}{q_{ij}} \frac{2 d_{ij} (1 + d_{ij}^2)^{-2}}{D} \tag{11.104}$$

As $\sum_{k=1}^{N} \sum_{l=1}^{N} p_{kl} = 1$, the second term becomes:

$$\frac{1}{D} \frac{\partial D}{\partial d_{ij}} = \frac{1}{D} \frac{\partial}{\partial d_{ij}} \left(\sum_{k=1}^{N} \sum_{l=1}^{N} (1 + d_{ij}^2)^{-1} \right) = \frac{2 d_{ij} (1 + d_{ij}^2)^{-2}}{D} \tag{11.105}$$

Now Eq. (11.103) can be written as:

$$\frac{\partial J}{\partial d_{ij}} = 2 d_{ij} \left(\frac{p_{ij}}{q_{ij}} - 1 \right) \frac{(1 + d_{ij}^2)^{-2}}{D} = 2 d_{ij} (p_{ij} - q_{ij}) (1 + d_{ij}^2)^{-1} \tag{11.106}$$

The last equality is due to Eq. (11.99).

The second derivative in Eq. (11.102) is:

$$\begin{aligned}\frac{\partial d_{ij}}{\partial \mathbf{y}_i} &= \frac{\partial}{\partial \mathbf{y}_i}\left[(\mathbf{y}_i-\mathbf{y}_j)^T(\mathbf{y}_i-\mathbf{y}_j)\right]^{1/2} \\ &= \frac{1}{2}\left[(\mathbf{y}_i-\mathbf{y}_j)^T(\mathbf{y}_i-\mathbf{y}_j)\right]^{-1/2} 2(\mathbf{y}_i-\mathbf{y}_j) = \frac{\mathbf{y}_i-\mathbf{y}_j}{d_{ij}}\end{aligned} \tag{11.107}$$

Finally, substituting Eqs. (11.106) and (11.107) into Eq. (11.102) we get the gradient of the objective function J with respect to each of the map points $\mathbf{y}_i$:

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{y}_i} &= 2\sum_{j=1}^{N}\frac{\partial J}{\partial d_{ij}}\frac{\partial d_{ij}}{\partial \mathbf{y}_i} = 4\sum_{j=1}^{N}(p_{ij}-q_{ij})(1+d_{ij}^2)^{-1}(\mathbf{y}_i-\mathbf{y}_j) \\ &= 4\sum_{j=1}^{N}\frac{p_{ij}-q_{ij}}{1+||\mathbf{y}_i-\mathbf{y}_j||^2}(\mathbf{y}_i-\mathbf{y}_j) = 4\sum_{j=1}^{N}w_{ij}(\mathbf{y}_i-\mathbf{y}_j) \\ &= \left(\sum_{j=1}^{N}w_{ij}\right)\mathbf{y}_i - \sum_{j=1}^{N}w_{ij}\mathbf{y}_j\end{aligned} \tag{11.108}$$

where $w_{ij} = (p_{ij}-q_{ij})/(1+d_{ij}^2)$ for all $i, j = 1, \cdots, N$. Having found the gradient vector, we can iteratively approach the optimal data points in the low-D space that minimize the discrepancy $J = D_{KL}(P||Q)$:

$$\mathbf{y}_{n+1} = \mathbf{y}_n - \delta\frac{\partial J}{\delta \mathbf{y}} + \alpha_n(\mathbf{y}_n - \mathbf{y}_{n-1}) \tag{11.109}$$

where δ is the step size (learning rate) and α is the coefficient for the momentum term in the direction of the previous iteration step (Eq. (2.56)).

The Metlab function below takes as input the data set $\mathbf{X}$ in the original high dimensional space, and generates as output the $N \times N$ matrix $\mathbf{P}$ of which the component p_{ij} in the ith row and jth column is defined in Eq. (11.98):

```
function P=data2prob(X)          % Given dataset X, find P matrix
    x2=sum(X.^2);                % squared distances between points
    D=x2+x2'-2*(X'*X);           % pair-wise squared distances
    N=size(D,1);                 % number of data points
    P=zeros(N);                  % initializing matrix P
    H0=3;                        % desired entropy value
    tol=1e-4;                    % tolerance
    for i=1:N                    % for each of the N data points
        d2=D(i,[1:i-1,i+1:N]); % squared distance between xi and others
        sigma=1;                 % default value of variance
        sigmamax=9*sigma;        % initial upper limit
        sigmamin=0;              % initial lower limit
        H=0;
        while abs(H-H0)>tol      % adjust sigma for desired entropy
            p=exp(-d2/sigma);    % Gaussian kernel
```

```
            p=p/sum(p);        % normalization
            H=-sum(p.*log(p)); % actual entropy based on current sigma
            if H<H0            % H too small
               sigmaamin=sigma; % update upper limit
               sigma=(sigma+sigmamax)/2; % increase sigma
            else               % H too small
               sigmamax=sigma; % update lower limit
               sigma=(sigma+sigmamin)/2; % reduce sigma
            end
        end
        P(i,[1:i-1,i+1:N])=p;  % Save resulting p for xi
    end
    P=(P+P')/2;                % make P symmetric
    P=P/sum(P(:));             % normalization
end
```

The Metlab function below takes as inputs matrix $\mathbf{P}$ generated by the previous function, *dim* as the dimension of the low dimensional space, and δ is the step size, and generates as output $\mathbf{Y}$ for the data points in the low dimensional space:

```
function Y=tSNE(P,dim,delta)
    N=size(P,1);
    Y=0.0001*randn(dim,N);   % random initialization of data
    dY=zeros(dim,N);         % increment for gradient descent
    alpha=0.5;               % coefficient for the momentum term
    it=0;
    while it<200
        it=it+1;
        Y0=Y;
        s=sum(Y.^2);         % squared distances between points
        T=1./(1+s+s'-2*(Y'*Y)); % t-distribution
        Q=T./sum(T(:));      % normalization
        W=(P-Q).*T;
        g=Y*(diag(sum(W,1))-W); % gradient of KL-divergence
        Y=Y-delta*g+alpha*dY % gradient descent with monumtum
        Y=Y-mean(Y,2);       % remove mean from dataset
        dY=Y-Y0;             % increment as momentum for next step
    end
end
```

The tSNE method is applied to the dataset of handwritten digits of 2240 256-D data points to visualize the dataset in 2-D and 3-D spaces as shown in Fig. 11.12. Comparing these results with those shown previously in Fig. 10.16, we see that the tSNE method is obviously more effective in terms of separating the ten classes in 2 or 3 out of 256 dimensional space than the PCA method.

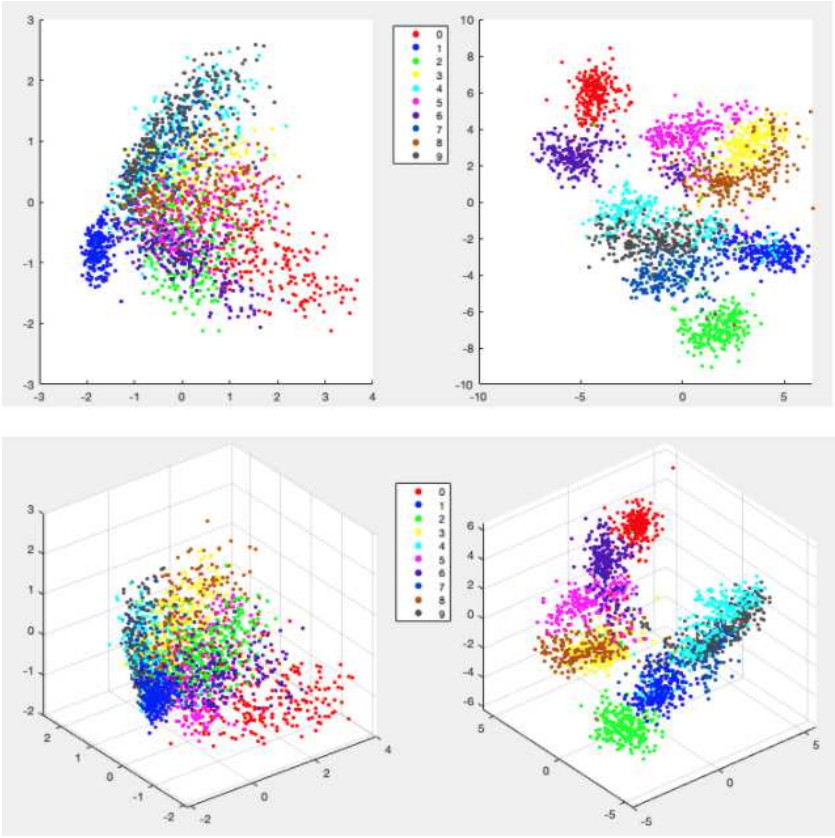


Figure 11.12 Visualization of 256-D dataset by the PCA (left) and tSNE (right) methods

Problems

1. Carry out kernel PCA to visualize the 4-D iris dataset in both 2-D and 3-D space, with all data points color coded according to their class identities. Compare your results with those obtained in the previous homework.
2. Carry out kernel PCA to visualize the 256-D handwritten digit dataset in both 2-D and 3-D space, with all data points color coded according to their class identities. Compare your results with those obtained in the previous homework.
3. Carry out probabilistic PCA to visualize the 256-D handwritten digit dataset in both 2-D and 3-D space, with all data points color coded according to their class identities. Compare your results with those obtained in the previous problem.
4. Use the PCoA method to reconstruct the locations of some cities in the US based on their pairwise distances, which can be obtained from their longitude-

latitude coordinates available here: https://www.latlong.net/category/cities-236-15.html or here: https://www.geodatos.net/en/coordinates/united-states. Compare your reconstructed map with the ground truth.

5. Apply PCoA method to visualize the iris dataset in both 2-D and 3-D spaces and compare the results with those obtained by the PCA method.
6. Apply PCoA method to visualize the handwritten digit dataset in both 2-D and 3-D spaces and compare the results with those obtained by the PCA method.
7. Apply the tSNE method to visualize the handwritten digit dataset in both 2-D and 3-D spaces and compare the results with those obtained by the PCA method.

12 Independent Component Analysis

The *Independent component analysis (ICA)* is a computational method widely used in signal processing and data mining to reveal the independent signal components hidden in an observed dataset. The methods of ICA and PCA considered in the previous chapters are similar in the sense that they both intend to find some essential signal components embedded in the data. However, the ICA is different from the PCA as it is based on a different assumption that each of the variables in the observed multivariate data is some linear combinations of a set of independent signal component. These independent components are to be separated based on a fundamental statistic property that the distribution of a linear combination of a set of random variables tends to be more Gaussian than those of the individual variables.

A typical problem that can be addressed by the ICA is the *blind source separation (BSS)* problem, for the purpose of separating and reconstructing a set of independent source signals from their linear mixtures. For example, in the "cocktail party problem", the goal is to reconstruct the individual voices of multiple speakers from a set of microphones recording all the conversations. The word "blind" means that no prior knowledge about either the sources or the mixing process is assumed to be available, except that the source signals are statistically independent. Although this BSS problem seems severely under constrained, the method of ICA can find nearly unique solutions satisfying certain properties.

In the following, we will first discuss the theoretical background of the ICA methods and then specifically consider two different ICA algorithms.

12.1 Independence and Non-Gaussianity

The goal of independent component analysis is to reconstruct a set of m independent source signals, called *independent components* and denoted by $\mathbf{s} = [s_1, \cdots, s_m]^T$, from another set of n observed variables denoted by $\mathbf{x} = [x_1, \cdots, x_n]^T$, each assumed to be a linear combination of the source signals:

$$\mathbf{x} = \mathbf{A}\mathbf{s} = \sum_{i=1}^{m} \mathbf{a}_i s_i \tag{12.1}$$

where $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_m]$ is an $n \times m$ *mixing matrix* of unknown values. We assume $n \geq m$ to avoid under determination. Given a dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ containing N observed data samples $\mathbf{x}_n = \mathbf{A}\mathbf{s}_n$ $(n = 1, \cdots, N)$, the ICA is to recover the independent source signals $\mathbf{s}_1, \cdots, \mathbf{s}_N$, based on a linear reconstruction model

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{W}\mathbf{A}\mathbf{s} \tag{12.2}$$

where $\mathbf{W}$ is a $m \times n$ reconstruction matrix, and $\mathbf{y} = [y_1, \cdots, y_m]^T$ contains m reconstructed signals $y_i = \mathbf{w}_i^T\mathbf{x}$ to match the corresponding independent signals s_i in $\mathbf{s}$, and $\mathbf{w}_i^T$ is the ith row vector of $\mathbf{W}$. It is obvious that for the reconstructed signal $\mathbf{y}$ to match exactly the source $\mathbf{s}$, ideally we desire to find $\mathbf{W} = \mathbf{A}^{-1}$. But as $\mathbf{A}$ is unknown, we need to find $\mathbf{W}$ in some different way, as the main task of ICA algorithms.

As ICA is an unsupervised method without labeled training data for the ground truth regarding either $\mathbf{A}$ or $\mathbf{s}$, we cannot find the reconstruction matrix $\mathbf{W}$ in closed form by the least squares method that minimizes the squared error $||\mathbf{y} - \mathbf{s}||^2$. Instead we have to find the optimal $\mathbf{W}$ as well as $\mathbf{s}$ based only on the requirement that the components in $\mathbf{y} = \mathbf{W}\mathbf{x}$ need to be as independent of each other as possible.

The method of ICA can be compared with the PCA. Both methods are to find a matrix $\mathbf{W}$ so that a given data vector $\mathbf{x}$ is mapped to another vector $\mathbf{y} = \mathbf{W}\mathbf{x}$. In PCA, the transform matrix $\mathbf{W} = \mathbf{V}^T$ (Eq. (10.20)) is the eigenvector matrix of the covariance matrix $\mathbf{\Sigma}_x$ of $\mathbf{x}$, i.e., the data point $\mathbf{x}$ is projected onto the eigenvectors $\mathbf{v}_i$ as the orthogonal basis of the space, and components $y_i = \mathbf{v}_i^T\mathbf{x}$ of $\mathbf{y}$ are completely decorrelated. (Only in the special case when $\mathbf{y} = [y_1, \cdots, y_n]$ is Gaussian, are its components also independent.) While in ICA, $\mathbf{W}$ is in general not orthogonal, and the components are independent and necessarily also uncorrelated. In this sense, the ICA is a more powerful method as it satisfies a stronger requirement for $\mathbf{W}$.

Here are some additional assumptions about the source signal $\mathbf{s}$ and the observed data $\mathbf{x} = \mathbf{A}\mathbf{s}$:

- The number n for the observed variables in $\mathbf{x}$ is no fewer than the number m for the independent source signals in $\mathbf{s}$. We assume $m = n$ in the following.
- The source signal $\mathbf{s}$ has a zero mean $\mathrm{E}[\mathbf{s}] = \mathbf{0}$ without loss of generality, and therefore so is the observed data: $\mathrm{E}[\mathbf{x}] = \mathrm{E}[\mathbf{A}\mathbf{s}] = \mathbf{A}\,\mathrm{E}[\mathbf{s}] = \mathbf{0}$.
- The source components are stochastically independent and therefore uncorrelated, and they have to be non-Gaussian for reasons to be discussed below.
- The estimation of $\mathbf{A}$ and $\mathbf{s}$ is up to a scaling factor. Let

$$\mathbf{C} = \mathrm{diag}(c_1, \cdots, c_n), \qquad \mathbf{C}^{-1} = \mathrm{diag}(1/c_1, \cdots, 1/c_n) \tag{12.3}$$

  and $\mathbf{A}' = \mathbf{A}\mathbf{C}^{-1}$ and $\mathbf{s}' = \mathbf{C}\mathbf{s}$, then we have

$$\mathbf{x} = \mathbf{A}\mathbf{s} = [\mathbf{A}\mathbf{C}^{-1}][\mathbf{C}\mathbf{s}] = \mathbf{A}'\mathbf{s}' \tag{12.4}$$

- Due to the arbitrary scaling factor, all independent components in $\mathbf{s}$ can be assumed to have unit variance $\sigma_i^2 = \mathrm{E}[s_i^2] = 1$. Also, as they are independent and necessarily uncorrelated, i.e.,

$$\sigma_{ij}^2 = \mathrm{E}[s_i s_j] = \delta_{ij}, \quad \text{i.e.} \quad E(\mathbf{s}\mathbf{s}^T) = \mathbf{I} \tag{12.5}$$

- The estimated independent components are not in any particular order. When the order of the corresponding elements in both $\mathbf{s}$ and $\mathbf{A}$ is rearranged, $\mathbf{x} = \mathbf{A}\mathbf{s}$ still holds.

The theoretical foundation of ICA algorithms is the *central limit theorem*, which states that the distribution of the linear combination of N independent random variables approaches Gaussian as $N \to \infty$. Specifically in a BSS problem, the observed variables in $\mathbf{x} = \mathbf{A}\mathbf{s}$ as a linear combination of the independent components $\{s_1, \cdots, s_m\}$ in $\mathbf{s}$ are necessarily more Gaussian than $\mathbf{s}$. To reconstruct $\mathbf{s}$ from $\mathbf{x}$, we need to reverse the process to reduce the Gaussianity. Based on this idea that *non-Gaussianity* is independence, the ICA method can be seen as the reverse process of the central limit theorem, with the goal of finding an optimal matrix $\mathbf{W}$ so that the components of the reconstructed $\mathbf{y} = \mathbf{W}\mathbf{x}$ are least Gaussian and therefore most independent.

The ICA can therefore be considered as an optimization problem for finding $\mathbf{W}$ that maximizes certain objective function that measures either the non-Gaussianity or the independence of $\mathbf{y} = \mathbf{W}\mathbf{x}$. We consider two main ICA algorithms below. The first one, called FastICA, maximizes the non-Gaussianity of $\mathbf{y}$, while the second one as a MLE method estimates matrix $\mathbf{A}$ by maximizing its likelihood, or equivalently the independence of the source signal $\mathbf{s}$ or the reconstructed signal $\mathbf{y}$. Due to their different criteria, these two algorithms may perform differently.

12.2 Preprocessing and Whitening

Before discussing the specific algorithms, we first consider the following operations as the preprocessing of the observed data so that all assumptions about the data $\mathbf{x}$ made above and required by the algorithm are satisfied.

- **Centering**

  Subtract $\mathrm{E}[\mathbf{x}]$ from $\mathbf{x}$ so that it becomes centered with zero mean. By doing so, we also have $\mathrm{E}[\mathbf{x}] = \mathrm{E}[\mathbf{A}\mathbf{s}] = \mathbf{A}\ \mathrm{E}[\mathbf{s}] = \mathbf{0}$, i.e., $E[\mathbf{s}] = \mathbf{A}^{-1}E[\mathbf{x}] = \mathbf{0}$.
- **Whitening**

  Find the eigenvalue matrix $\mathbf{\Lambda}$ and eigenvector matrix $\mathbf{V}$ of the covariance matrix $E[\mathbf{x}\mathbf{x}^T]$ so that

$$\mathbf{V}^T E[\mathbf{x}\mathbf{x}^T]\mathbf{V} = \mathbf{\Lambda}, \qquad \text{i.e.,} \quad \mathbf{\Lambda}^{-1/2}\mathbf{V}^T E[\mathbf{x}\mathbf{x}^T]\mathbf{V}\mathbf{\Lambda}^{-1/2} = \mathbf{I} \tag{12.6}$$

As $E[\mathbf{x}\mathbf{x}^T]$ is symmetric, $\mathbf{V}^T = \mathbf{V}^{-1}$ is orthogonal. Then carry out a linear transform

$$\mathbf{x}' = (\mathbf{\Lambda}^{-1/2}\mathbf{V}^T)\mathbf{x}, \qquad \text{i.e.,} \quad \mathbf{x} = (\mathbf{\Lambda}^{-1/2}\mathbf{V}^T)^{-1}\mathbf{x}' = (\mathbf{V}\mathbf{\Lambda}^{1/2})\mathbf{x}' \tag{12.7}$$

so that the components of the data are not only decorrelated (same as the KLT transform) but also normalized with unity variance:

$$\begin{aligned}\mathrm{E}[\mathbf{x}'\mathbf{x}'^T] &= \mathrm{E}[(\mathbf{\Lambda}^{-1/2}\mathbf{V}^T\mathbf{x})(\mathbf{\Lambda}^{-1/2}\mathbf{V}^T\mathbf{x})^T] = \mathrm{E}[\mathbf{\Lambda}^{-1/2}\mathbf{V}^T\mathbf{x}\mathbf{x}^T\mathbf{V}\mathbf{\Lambda}^{-1/2}] \\ &= \mathbf{\Lambda}^{-1/2}\mathbf{V}^T\,\mathrm{E}[\mathbf{x}\mathbf{x}^T]\mathbf{V}\mathbf{\Lambda}^{-1/2} = \mathbf{I}\end{aligned} \tag{12.8}$$

i.e., $\mathbf{x}'$ is whitened with covariance $\sigma_{ij}^2 = \delta_{ij}$.

Once $\mathbf{x}$ is centered and whitened to become $\mathbf{x}' = (\mathbf{\Lambda}^{-1/2}\mathbf{V}^T)\mathbf{x}$, the reconstruction becomes

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{W}(\mathbf{V}\mathbf{\Lambda}^{1/2})\mathbf{x}' = \mathbf{W}'\mathbf{x}' \tag{12.9}$$

where $\mathbf{W}' = \mathbf{W}\mathbf{V}\mathbf{\Lambda}^{1/2}$ is the reconstruction matrix for whitened $\mathbf{x}'$. We also have

$$\mathrm{E}[\mathbf{x}'\mathbf{x}'^T] = \mathrm{E}[(\mathbf{A}\mathbf{s})(\mathbf{A}\mathbf{s})^T] = \mathbf{A}\,\mathrm{E}[\mathbf{s}\mathbf{s}^T]\mathbf{A}^T = \mathbf{A}\mathbf{A}^T = \mathbf{I} \tag{12.10}$$

As $\mathbf{y} = \mathbf{W}'\mathbf{x}'$ is to match $\mathbf{s}$, we assume they have the same covariance $\mathrm{E}[\mathbf{y}\mathbf{y}^T] = \mathrm{E}[\mathbf{s}\mathbf{s}^T] = \mathbf{I}$ (Eq. (12.5)), and further get

$$\mathbf{I} = \mathrm{E}[\mathbf{y}\mathbf{y}^T] = \mathrm{E}[(\mathbf{W}'\mathbf{x}')\,(\mathbf{W}'\mathbf{x}')^T] = \mathbf{W}'\,\mathrm{E}[\mathbf{x}'\mathbf{x}'^T]\mathbf{W}'^T = \mathbf{W}'\mathbf{W}'^T \tag{12.11}$$

We see that after centering and whitening, both the mixing matrix $\mathbf{A}$ and the reconstruction matrix $\mathbf{W}'$ becomes orthogonal and can therefore be interpreted as rotation matrices. For convenience, in the following discussion we will still denote the whitened data samples by $\mathbf{x}$ and the reconstruction matrix by $\mathbf{W}$.

12.3 Non-Gaussianity and FastICA

In ICA, we desire to find the optimal $\mathbf{W}$ that maximizes the non-Gaussianity of the reconstructed $\mathbf{y} = \mathbf{W}\mathbf{x}$. In general, the non-Gaussianity of a random variable y with a density function $p(y)$ can be measured by

- **Kurtosis**

  The excess kurtosis (Section B.2.4) of the centered and whitened random variable x (with $\mu_x = 0$ and $\sigma_x = 1$) is defined as:

  $$\mathrm{Kurt}(x) = E\{x^4\} - 3 \tag{12.12}$$

  which is zero only when x has a Gaussian distribution. A non-Gaussian distribution is either supergaussian if its kurtosis is positive or subgaussian if its kurtosis is negative. Therefore the absolute value of the excess kurtosis or kurtosis squared can be used to measure the non-Gaussianity of a distribution. However, as kurtosis is very sensitive to noisy outliers, it is not a robust measurement of non-Gaussianity.

- **Negentropy**

  The negentropy of a random variable y is defined as the difference between its entropy $H(y)$ (Section B.2.1) and the entropy of a Gaussian variable g with the same variance σ^2:

$$J(y) = H(g) - H(y) = C - H(y) \geq 0 \tag{12.13}$$

  where $C = H(g) = \log(2\pi e\sigma^2)/2$ ((Eq. (B.183)) is a constant independent of y. Here $J(y)$ is non-negative as Gaussian distribution has the maximum entropy among all distributions with the same variance. Negentropy $J(y)$ measures non-Gaussianity as it takes a larger value when y is less Gaussian. The negentropy defined for a scalar random variable y can be generalized to a random vector $\mathbf{y} = [y_1, \cdots, y_m]^T$:

$$J(\mathbf{y}) = H(\mathbf{y}_G) - H(\mathbf{y}) = C - H(\mathbf{y}) \geq 0 \tag{12.14}$$

While kurtosis and negentropy of a random variable y are good measurement for the non-Gaussianity of y, they both depend on its distribution $p(y)$, which is typically not readily available, and the negentropy $J(\mathbf{y})$ as theoretically defined above can only be approximated as below based on the maximum-entropy principle:

$$J(y) \approx \sum_i c_i(\mathrm{E}[G_i(y)] - \mathrm{E}[G_i(g)])^2 \tag{12.15}$$

where g is again a Gaussian variable and both y and g are assumed to have zero mean and unit variance, and G_i is some non-quadratic function such as the following also shown in in Fig. 12.1:

$$G_1(y) = \frac{1}{a}\log\,\cosh\,(a\,y), \qquad G_2(y) = -\exp(-y^2/2) \tag{12.16}$$

where $1 \leq a \leq 2$ is a constant. This approximated negentropy is always greater than zero except when x is Gaussian. In particular, when only a single term is used in Eq. (12.15), we have

$$J(y) \approx (\mathrm{E}[G(y)] - \mathrm{E}[G(g)])^2 \tag{12.17}$$

Since the second term is a constant, we want to maximize $\mathrm{E}\{G(y)\}$ to maximize $J(y)$.

Based on this approximated negentropy, the FastICA algorithm finds the reconstruction matrix $\mathbf{W}$ that maximizes the sum of the approximated negentropies of all m components of the reconstructed signal $\mathbf{y} = \mathbf{W}\mathbf{x}$:

$$\sum_{i=1}^{m} \mathrm{E}[G(y_i)] = \sum_{i=1}^{m} \mathrm{E}[G(\mathbf{w}_i^T\mathbf{x})] \tag{12.18}$$

where $y_i = \mathbf{w}_i^T\mathbf{x}$ is the ith component of $\mathbf{y}$ and $\mathbf{w_i}^T$ is the ith row vector of $\mathbf{W}$.

We first consider computing $\mathbf{w}_i$ for y_i as one of the m components (with the subscript i dropped for now). Specifically, we need to find a vector $\mathbf{w}$ that

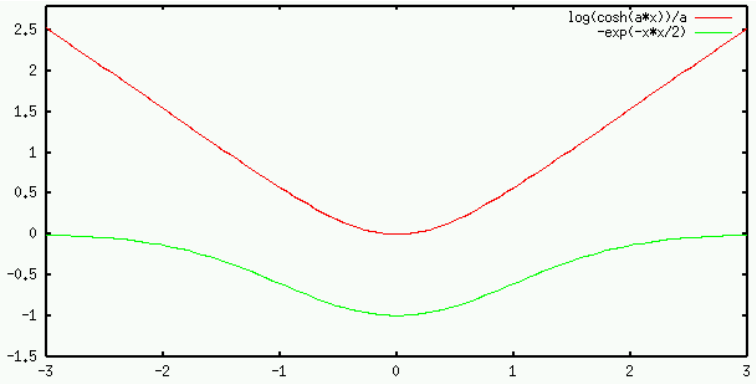


Figure 12.1 Two typical G-functions

maximizes $\mathrm{E}[G(y)]$ as the objective function subject to the constraint that $\mathbf{w}$ has to be normalized, i.e., $\mathbf{w}^T\mathbf{w} = 1$. To solve this constrained optimization problem we first construct the Lagrangian function:

$$L(\mathbf{w}) = \mathrm{E}[G(\mathbf{w}^T\mathbf{x})] - \frac{\lambda}{2}(\mathbf{w}^T\mathbf{w} - 1) \tag{12.19}$$

and then set to zero its derivative with respect to $\mathbf{w}$:

$$\mathbf{g}_L(\mathbf{w}) = \frac{d}{d\mathbf{w}}L(\mathbf{w}) = \mathrm{E}[\mathbf{x}\,\frac{d}{d\mathbf{w}}G(\mathbf{w}^T\mathbf{x})] - \lambda\mathbf{w} = \mathrm{E}[\mathbf{x}\,G'(\mathbf{w}^T\mathbf{x})] - \lambda\mathbf{w} = 0 \tag{12.20}$$

where $G'(z) = dG(z)/dz$ is the derivative of function $G(z)$. This equation system can be solved for $\mathbf{w}$ iteratively by the Newton-Raphson method:

$$\mathbf{w} \Leftarrow \mathbf{w} - \mathbf{J}_g^{-1}(\mathbf{w})\mathbf{g}(\mathbf{w}) \tag{12.21}$$

Here $\mathbf{J}_f(\mathbf{w})$ is the Jacobian matrix of $\mathbf{g}(\mathbf{w}) = dL(\mathbf{w})/d\mathbf{w}$:

$$\mathbf{J}_g(\mathbf{w}) = \frac{d}{d\mathbf{w}}\mathbf{g}(\mathbf{w}) = \mathrm{E}[\mathbf{x}\mathbf{x}^T G''(\mathbf{w}^T\mathbf{x})] - \lambda\mathbf{I} \tag{12.22}$$

where $G''(z) = d^2G(z)/d^2z$ is the second order derivative of $G(z)$. We further approximate the first term on the right to get

$$\mathrm{E}[\mathbf{x}\mathbf{x}^T G''(\mathbf{w}^T\mathbf{x})] \approx \mathrm{E}[\mathbf{x}\mathbf{x}^T]\ \mathrm{E}[G''(\mathbf{w}^T\mathbf{x})] = \mathrm{E}[G''(\mathbf{w}^T\mathbf{x})]\mathbf{I} \tag{12.23}$$

where $\mathrm{E}[\mathbf{x}\mathbf{x}^T] = \mathbf{I}$ as the dataset is assumed to be whitened, then the Jacobian becomes diagonal

$$J_f(\mathbf{w}) = \left(\mathrm{E}[G''(\mathbf{w}^T\mathbf{x})] - \lambda\right)\mathbf{I} \tag{12.24}$$

and the Newton-Raphson iteration becomes:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \frac{1}{\mathrm{E}[G''(\mathbf{w}_n^T\mathbf{x})] - \lambda}\left(\mathrm{E}[\mathbf{x}\,G'(\mathbf{w}_n^T\mathbf{x})] - \lambda\mathbf{w}\right) \tag{12.25}$$

Multiplying both sides by $c = \lambda - \mathrm{E}[G''(\mathbf{w}_n^T\mathbf{x})]$, we get

$$c\mathbf{w}_{n+1} = \mathrm{E}[\mathbf{x}G'(\mathbf{w}_n^T\mathbf{x})] - \mathrm{E}[G''(\mathbf{w}_n^T\mathbf{x})]\mathbf{w} \tag{12.26}$$

where the scaling factor c can be removed as the updated $\mathbf{w}$ will be normalized.

This algorithm can then be repeated to get all row vectors in $\mathbf{W}$ each for one of the m row vectors $\mathbf{w}_i$. However, if this process is carried out independently, the resulting vectors may not be orthogonal to each other as required. To address this issue, the Gram-Schmidt method in Section A.1.2 can be used. Specifically, during each iteration of the computation for $\mathbf{w}_i$ for $i > 1$, before it is normalized, we also subtract its projection onto each of the previously found $\mathbf{w}_1, \cdots, \mathbf{w}_{i-1}$ (Eq. (A.39)), so that it is guaranteed to be orthogonal to all of them. All vectors such computed will guarantee that the final matrix $\mathbf{W}$ is orthogonal satisfying $\mathbf{W}^T = \mathbf{W}^{-1}$ as required.

Now we have the following FastICA algorithm that is carried out m times each for one of the vectors $\mathbf{w}_i$, $(i = 1, \cdots, m)$:

1. Choose an initial random guess for $\mathbf{w}_i$
2. Iterate:

$$\mathbf{w} = \mathrm{E}[\mathbf{x}G'(\mathbf{w}^T\mathbf{x})] - \mathrm{E}[G''(\mathbf{w}^T\mathbf{x})]\mathbf{w} \tag{12.27}$$

   where the expectation is estimated as the average of all N samples in the dataset.
3.

$$\mathbf{w} \longleftarrow \mathbf{w} - \sum_{j=1}^{i-1}(\mathbf{w}^T\mathbf{w}_j)\mathbf{w}_j \tag{12.28}$$

4. Normalize:

$$\mathbf{w} \longleftarrow \mathbf{w}/||\mathbf{w}|| \tag{12.29}$$

5. If not converged, go back to step 2.

The Metlab code segment below shows the essential part of the FastICA algorithm, in which the reconstruction matrix $\mathbf{W}$ is calculated based on dataset $\mathbf{X}$ containing N observed data samples. Here`Imax` is the maximum number of iterations and `tol` is the tolerance, e.g., 10^{-6}.

```
syms u
G=-exp(-u^2/2);
G=log(cosh(u));              % symbolic function G
g=matlabFunction(diff(G));   % G'
dg=matlabFunction(diff(diff(G))); % G''
[d N]=size(X);               % size of dataset
[X T]=whiten(X);             % function that whitens dataset
W=[];                        % initialize weight matrix W
for i=1:d                    % for each weight vector w
```

```
    w=rand(d,1);              % random initialization
    for i=1:Imax
        wold=w;               % old weight vector
        wx=w'*X;
        w=X*g(wx)'-sum(dg(wx))*w; % new weight vector
        if i>1
            w=w-W*W'*w;       % Gram-Schmidt process
        end
        w=w/norm(w);          % renormalization
        if norm(w-wold)<tol % convergence
            break
        end
    end
    W=[W w];
end
Y=W'*X;                       % signal Reconstruction by ICA
W=W*T;
```

Also, listed below is the whitening function `whiten` used in the code above that takes the dataset `X` as input and generates the centered and whitened data and the whitening transform matrix `T` as the output:

```
function [X T]=whitening(X)
    [d N]=size(X);
    X=X-repmat(mean(X,2),1,N);
    [U S V]=svd(X);
    T=sqrt(N)*diag(1./(diag(S(1:d,1:d))))*U';
    X=T*X;
end
```

12.4 Likelihood and Independence Maximization

The ideal reconstruction matrix $\mathbf{W}$ can also be estimated by solving an optimization problem based on the following different but equivalent criteria all related to the assumed independence of the source signal $\mathbf{s}$, or, equivalently, the desired independence of the reconstructed signal $\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{W}\mathbf{A}\mathbf{s}$.

- The optimal $\mathbf{W}$ can be found so that the reconstructed signal $\mathbf{y} = \mathbf{W}\mathbf{x}$ is maximally independent measured by the mutual information (Eq. (B.201))

shared by its components $\{y_1, \cdots, y_m\}$:

$$\begin{aligned} I(\mathbf{y}) &= \sum_{i=1}^{n} H(y_i) - H(\mathbf{y}) = -\sum_{i=1}^{n} \mathrm{E}\left[\log p(y_i)\right] - H(\mathbf{W}\mathbf{x}) \\ &= -\mathrm{E}\left[\sum_{i=1}^{n} \log p(y_i)\right] - \log\ |det\, \mathbf{W}| - H(\mathbf{x}) \end{aligned} \tag{12.30}$$

where $H(\mathbf{y}) = H(\mathbf{W}\mathbf{x}) = H(\mathbf{x}) + \log\ |\det\ \mathbf{W}|$ (Eq. (B.188)), and the last term $H(\mathbf{x})$ is unrelated to $\mathbf{W}$ and can therefore be dropped.

- Matrix $\mathbf{A}$ can be estimated so that $p(\mathbf{x})$ of the observed data $\mathbf{x}$ most closely matches $p(\mathbf{x}|\mathbf{A})$ based on the assumed independence of the components in the source $\mathbf{s}$ (Eq. (B.85)):

$$p(\mathbf{x}|\mathbf{A}) = p(\mathbf{A}\mathbf{s}) = |\det \mathbf{A}^{-1}|\, p(\mathbf{s}) = |\det \mathbf{A}^{-1}| \prod_{i=1}^{m} p_i(s_i) \tag{12.31}$$

Now the optimal $\mathbf{A}$ can be found as the one that minimizes the KL-divergence between $p(\mathbf{x}$ and $p(\mathbf{x}|\mathbf{A})$:

$$\begin{aligned} & D_{KL}(p(\mathbf{x})||p(\mathbf{x}|\mathbf{A}) = \int p(\mathbf{x}) \log \frac{p(\mathbf{x})}{p(\mathbf{x}|\mathbf{A})} d\mathbf{x} \\ &= \int p(\mathbf{x}) \log p(\mathbf{x}) d\mathbf{x} - \int p(\mathbf{x}) \log \left(|\det \mathbf{A}^{-1}| \prod_{i=1}^{m} p_i(s_i) \right) d\mathbf{x} \\ &= H(\mathbf{x}) - \log |\det \mathbf{A}^{-1}| \int p(\mathbf{x}) d\mathbf{x} - \int p(\mathbf{x}) \left(\sum_{i=1}^{m} \log p_i(s_i) \right) d\mathbf{x} \\ &= H(\mathbf{x}) - \log |\det \mathbf{A}^{-1}| - E_x \left[\sum_{i=1}^{m} \log p_i(s_i) \right] \end{aligned} \tag{12.32}$$

Here the first term $H(\mathbf{x})$ independent of $\mathbf{A}$ can be dropped, and the expectation with respect to the observed data $\mathbf{x}$ in the last term can be estimated as the average of the all samples in the dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$.

- Matrix $\mathbf{A}$ can also be estimated by maximizing its likelihood given the observed dataset $\mathbf{X}$ containing N i.i.d. samples:

$$\begin{aligned} L(\mathbf{A}|\mathbf{X}) = p(\mathbf{X}|\mathbf{A}) &= \prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{A}) = \prod_{n=1}^{N} |\det \mathbf{A}^{-1}| \prod_{i=1}^{m} p_i(s_{in}) \\ &= |\det \mathbf{A}^{-1}|^N \prod_{n=1}^{N} \prod_{i=1}^{m} p_i(s_{in}) \end{aligned} \tag{12.33}$$

where $p(\mathbf{x}_n|\mathbf{A})$ is given in Eq. (12.31) and s_{in} is the ith component of $\mathbf{s}_n$.

We further get the log-likelihood:

$$\begin{aligned} l(\mathbf{A}|\mathbf{X}) = \log L(\mathbf{A}|\mathbf{X}) &= \log \left[|\det \mathbf{A}^{-1}|^N \prod_{n=1}^{N} \prod_{i=1}^{m} p_i(s_{in}) \right] \\ &= N \left[\log |\det \mathbf{A}^{-1}| + \frac{1}{N} \sum_{n=1}^{N} \sum_{i=1}^{m} p_i(s_{in}) \right] \\ &\propto \log |\det \mathbf{A}^{-1}| + E_x \left[\sum_{i=1}^{m} \log p_i(s_i) \right] \end{aligned} \tag{12.34}$$

Here the average over all N data samples in the second term is treated as the estimated expectation with respect to $\mathbf{x}$.

All three cases above are equivalent, as maximizing the log-likelihood in Eq. (12.34) is equivalent to minimizing mutual information of either the reconstructed signal $\mathbf{y}$ or the source signal $\mathbf{s}$ in Eq. (12.30) or (12.32), thereby maximizing their independence.

Also, we see that for the reconstructed signal $\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{W}\mathbf{A}\mathbf{s}$ to match perfectly the source signal $\mathbf{s}$, the reconstruction matrix simply needs to be $\mathbf{W} = \mathbf{A}^{-1}$ so that $y_i = \mathbf{w}_i^T \mathbf{x} = s_i$ for all $i = 1, \cdots, m$, and the three criteria above become exactly the same. We can now find the optimal $\mathbf{W}$ that maximizes Eq. (12.34) as the objective function:

$$J(\mathbf{W}) = \log |\det \mathbf{W}| + E \left[\sum_{i=1}^{m} \log p_i(\mathbf{w}_i^T \mathbf{x}) \right] \tag{12.35}$$

The gradient of this objective function is:

$$\mathbf{g}_J(\mathbf{W}) = \frac{d}{d\mathbf{W}} J(\mathbf{W}) = \frac{d}{d\mathbf{W}} \log |\det \mathbf{W}| + E \left[\frac{d}{d\mathbf{W}} \sum_{i=1}^{m} \log p_i(\mathbf{w}_i^T \mathbf{x}) \right] \tag{12.36}$$

The first term is (Eq. (B.245)):

$$\frac{d}{d\mathbf{W}} \log |\det \mathbf{W}| = (\mathbf{W}^{-1})^T = (\mathbf{W}^T)^{-1} \tag{12.37}$$

For each term in the summation of the second term, $\mathbf{w}_i^T \mathbf{x} = y_i$ is the ith component of $\mathbf{y}$, and the derivative with respect to the kth component w_{ki} of $\mathbf{w}_i$ is:

$$\frac{d}{d\, w_{ki}} \log p_i(\mathbf{w}_i^T \mathbf{x}) = \frac{d}{d\, w_{ki}} \log p_i(y_i) = \frac{p_i'(y_i)}{p_i(y_i)}\, x_k = \phi_i x_k \tag{12.38}$$

where we have defined

$$\phi_i = \phi(y_i) = \frac{p'(y_i)}{p(y_i)} \tag{12.39}$$

If we further assume specifically $p(y) = 1/\cosh(y)$ as a non-Gaussian distribution, and get

$$\phi(y) = \frac{(1/\cosh(y))'}{1/\cosh(y)} = -\frac{\sinh(y)}{\cosh(y)} = -\tanh(y) \tag{12.40}$$

then the second term of Eq. (12.36) can be written as $\mathrm{E}[\boldsymbol{\phi}(\mathbf{y})\mathbf{x}^T]$, where $\boldsymbol{\phi}(\mathbf{y}) = [\phi_1, \cdots, \phi_m]^T$, and the gradient becomes:

$$\mathbf{g}_J(W) = (\mathbf{W}^{-1})^T + \mathrm{E}[\boldsymbol{\phi}\mathbf{x}^T] \tag{12.41}$$

Now the optimal $\mathbf{W}$ that maximizes $J(\mathbf{W})$ can be found iteratively based on the gradient ascend method:

$$\mathbf{W}_{n+1} = \mathbf{W}_n + \delta \mathbf{g}_J(\mathbf{W}_n) = \mathbf{W}_n + \delta \left[(\mathbf{W}_n^{-1})^T + \mathrm{E}[\boldsymbol{\phi}(\mathbf{y})\mathbf{x}^T]\right] \tag{12.42}$$

where δ is the step size, and the expectation can be estimated as the average of all N data samples.

The gradient $\mathbf{g}_J(\mathbf{W})$ can be modified by post-multiplying $\mathbf{W}^T\mathbf{W}$ to get the *natural gradient* (Section B.3.4):

$$\left[(\mathbf{W}^T)^{-1} + \mathrm{E}[\boldsymbol{\phi}\mathbf{x}^T]\right] \mathbf{W}^T\mathbf{W} = \mathbf{W} + \mathrm{E}[\boldsymbol{\phi}\, \mathbf{x}^T\mathbf{W}^T\, \mathbf{W}] = \mathbf{W} + \mathrm{E}[\boldsymbol{\phi}\mathbf{y}^T\mathbf{W}] \tag{12.43}$$

The last equality is due to $\mathbf{W}\mathbf{x} = \mathbf{y}$. Now the iteration becomes:

$$\mathbf{W}_{n+1} = \mathbf{W}_n + \delta \mathbf{g}_J(\mathbf{W}_n) = \mathbf{W}_n + \delta \left(\mathbf{W}_n + \mathrm{E}[\boldsymbol{\phi}\mathbf{y}^T\mathbf{W}_n]\right) \tag{12.44}$$

Based this natural gradient along a modified search direction in each step of the iteration, it will converge more quickly.

The Metlab code segment listed below shows the essential part of the MLE algorithm, where `Imax` is the maximum number of iterations, `\lambda` is the step size, and `tol` is the tolerance, e.g., 10^{-6}. In the algorithm, data preprocessing (centering and whitening) is not required and is therefore optional. Here matrix $\mathbf{W}$ can be updated based on either the gradient or the natural gradient which converges much more quickly.

```
    [d N]=size(X);      % size of dataset
    T=eye(d);           % default whitening transform matrix
    if preprocessing    % optional preprocessing
        [X T]=whiten(X);
    end
    W=eye(d);           % initialize W
    lambda=0.5;         % step size
    tol=10^(-6);        % tolerance
    for i=1:Imax
        Y=W*X;          % reconstruction of y
%        Wnew=W+lambda*(inv(B)'-tanh(Y)*X'/N);
                                            % update W by gradient
        Wnew=W+lambda*(W-tanh(Y)*(Y'*W)/N); % or natural gradient
        if mean(abs(Wnew(:)-W(:))) < tol    % check convergence
            break
        end
        W=Wnew;
```

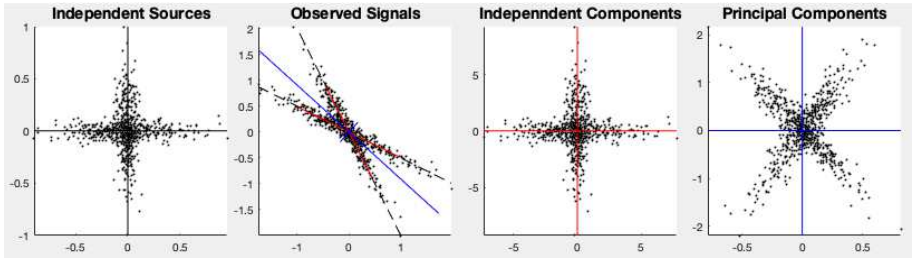


Figure 12.2 Reconstruction of independent components by ICA based on ML (2D)

```
end
W=W*T;
```

Example 12.1 This example demonstrates the ICA algorithm based on the maximum likelihood method for the reconstruction of two independent source signals in $\mathbf{s} = [s_1, s_2]^T$, shown in the first panel in Fig. 12.2, based on the observed data $\mathbf{x} = [x_1, x_2]^T = \mathbf{A}_0\mathbf{s}$, shown in the second panel in the figure.

The two reconstructed independent components by the ICA algorithm are shown in the third panel, which match the ground truth source signals in the first panel very well.

Also the estimated mixing matrix $\mathbf{A}$ is listed below, together with the ground truth $\mathbf{A}_0$ (not used in the algorithm):

$$\mathbf{A} = \begin{bmatrix} 0.243 & -0.111 \\ -0.120 & 0.218 \end{bmatrix}, \qquad \mathbf{A}_0 = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \tag{12.45}$$

We see that the ground truth $\mathbf{A}_0$ is matched by the estimated $\mathbf{A}$ very closely, when scaled up by a factor of 9.

In the second panel, we also plotted the column vectors of the ground truth mixing matrix $\mathbf{A}_0$ (black dashed lines), together with those of the estimated mixing matrix $\mathbf{A}$ (red lines). These two sets of lines match each other very closely, both representing the two independent sources.

For comparison, the PCA method is also used to find the principal components of the dataset, as shown by the blue lines in the fourth panel. They are also plotted in the second panel, as the eigenvectors of the covariance matrix of the dataset, weighted by the corresponding eigenvalues. As expected, these eigenvectors are along the major and minor axes of an ellipse approximating the overall distribution of observed data points. We see that the principal components are very different from the independent components..

The same algorithm is also applied to a 3-D dataset and the results are shown in Fig. 12.3.

Example 12.2 This example illustrates the results of the FastICA algorithm

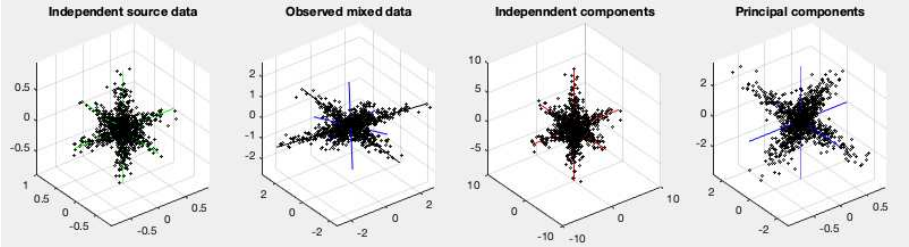


Figure 12.3 Reconstruction of independent components by ICA based on ML (3D)

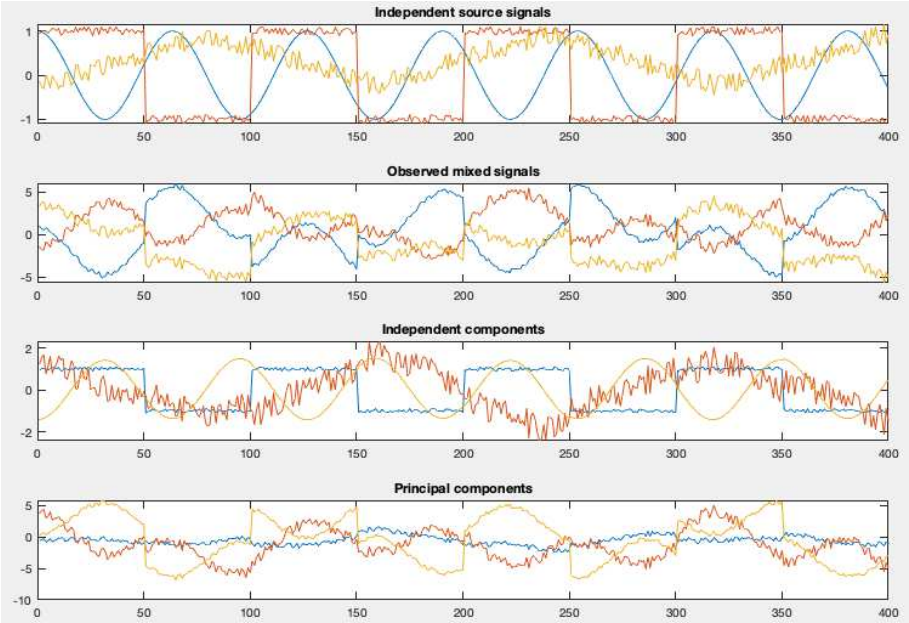


Figure 12.4 Reconstruction of independent components in time signals by FastICA

when applied to reconstruct the independent components of three observed time signals shown in the 2nd panel of Fig. 12.4, each as a linear combination of some three independent source signals (noisy square wave, triangular wave, and sinusoidal wave) as shown in the first panel.

The reconstructed independent components are shown in the third panel, which closely match the ground truth source signals (first panel), up to some scaling factors (could be negative). For comparison, the principal components obtained by the PCA method are also shown in the fourth panel, which are very different from the desired independent components.

Problems

1. Implement the ICA algorithm based on maximum likelihood method and apply your code to both the 2D and 3D datasets used in Example 12.1.

   The independent source signals can be generated by the code below:

```
function S=data(d,N)
    c=30;                    % variance of independent signals
    if d==2                  % 2D signals
        m=[0 0]';            % zero mean
        C1=[1 0; 0 c];
        C2=[c 0; 0 1];       % covariance of two independent sets
        S=mvnrnd(m,C1,N)';   % generate 2 sets of N data points
        S=[S mvnrnd(m,C2,N)']; % with covariances C1 and C2
    elseif d==3              % 3D signals
        m=[0 0 0]';          % zero mean
        C1=eye(3);  C2=C1;  C3=C1;
        C1(1,1)=c;  C2(2,2)=c;  C3(3,3)=c;
                             % covariance of three independent sets
        S=mvnrnd(m,C1,N)';   % generate 3 sets of N data points
        S=[S mvnrnd(m,C2,N)']; % with covariances C1, C2 and C3
        S=[S mvnrnd(m,C3,N)'];
    end
    S = S./max(abs(S(:)));   % normailization
end
```

   The observed data can then be obtained by linearly combining the independent source signals:

```
    if d==2
        A=[2 -1; -1 2];                % 2D mixing matrix
    elseif d==3
        A=[3 -2 1; -2 1 3; 1 2 -3];    % 3D mixing matrix
    end
    X=A*S;     % generate observed data by mixing independent sources
```

2. Implement the algorithm of FastICA and apply your code to some observed time signals each as a linear combination of some independent components (such as square, sinusoidal, and triangular waves of different frequencies) contaminated by some random noise, similar to those used in Example 12.4.

Part IV

Classification

In Part IV, we consider *pattern classification*. As one of the most important topics in machine learning, classification can be considered as a learning process that finds out the underneath structure of the given dataset. Many real world problems can be formulated as to classify all data points in the given dataset into different categories. For example, in image recognition, the given data may be a set of images of various objects of interest (e.g., different animals or handwritten alphanumeric symbols), and the goal is to recognize these objects by classifying them into different categories each for one type of the objects. Such objects of interest are generally referred to as *patterns*, and the categories are the *classes*. In such cases, pattern classification is also called *pattern (or object) recognition*.

A classification problem can be generally formulated as to classify a set of N data samples, each represented as a vector $\mathbf{x} = [x_1, \cdots, x_d]^T$, a point in the d-dimensional feature space, into a set of K classes denoted by $\{C_1, \cdots, C_K\}$, in such a way that all similar points are classified into the same class.

In the context of probabilistic methods of classification, these data points are assumed to be independent samples taken from a population with the same probability distribution, i.e., they can be treated as a set of independent and identically distributed (i.i.d.) samples of the population. Geometrically, the task of classifying similar data samples into the same classes can be considered as to partition the feature space into a set of regions each corresponding to one of the K classes. The boundary between any two such regions is called the *decision boundary*, and the goal of a specific classification algorithm, called a classifier, is to determine these decision boundaries based on the given dataset.

Classification is a *supervised* learning process if each sample $\mathbf{x}_n$ in the given dataset is associated with some labeling y_n, representing some prior knowledge of the data, such as the class identities of the samples. For example, we can use $y_n \in \{1, \cdots, N\}$ to indicate sample $\mathbf{x}_n$ is known to belong to class C_{y_n}, or in the case of binary classification, we can use either $y_n \in \{0,\ 1\}$ or $y_n \in \{-1,\ 1\}$ to indicate $\mathbf{x}$ belongs to either of the two classes. How the samples are labeled depends on the specific classification algorithms.

Such a set of labeled data samples is called the *training set* and represented by $\mathcal{D} = \{(\mathbf{x}_n,\ y_n)\ (n = 1, \cdots, N)\}$, which can also be denoted by $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$, where $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ is $d \times N$ array containing all N samples, and $\mathbf{y} = [y_1, \cdots, y_N]^T$ is an N-D vector containing their corresponding labelings. We can further denote by N_k the number of samples all labeled to belong to class C_k, so that $N = \sum_{k=1}^{K} N_k$.

Supervised classification is a two-stage process, in which a classifier is first constructed by learning the training set in the training stage, and then tested in the testing stage where any unlabeled data samples can be classified into one of the classes. We will consider different types of classification algorithms in Chapters 13 and 14.

If the samples in $\mathbf{X}$ are not labeled, i.e., $\mathbf{y}$ is unavailable in the given dataset $\mathcal{D}$, then classification becomes an *unsupervised* learning process, called *clustering*, and the task becomes to group or cluster all data points into a set of clusters

in such a way that similar data points (close to each other in the feature space) are in the same cluster while dissimular ones (far away from each other) are in different clusters. Unsupervised clustering is similar to supervised classification in the sense that they both partition the feature space into different regions each for one of the classes/clusters containing similar data samples, but clustering is very different from classification in the sense that the partitioning is totally based on how the data points are distributed in the feature space, without any prior knowledge, such as the labeling. We will consider various unsupervised clustering algorithms in Chapter 15.

Also in Chapter 15, we will consider a set of unsupervised algorithms for data clustering without the training set containing labeled data samples.

A wide variety of supervised and unsupervised learning algorithms will be discussed in the chapters of this Part, as well as those in Part V (for neural network methods), all for the same purpose of classification or clustering. It is natural to ask then, why do we need so many different algorithms? which of these algorithms is the best of them all? The answer is it depends. Every learning algorithm may be better suited for a certain type of learning problems than others, depending on many specifics such as the computational cost, the size of the dataset, how noisy or reliable the dataset is, how separable the classes are, etc. Moreover, even for the same learning algorithm, there are typically many hyperparameters to be adjusted in order to achive the optimal performance on a specific dataset. Therefore the reader is encouraged to experiment with different settings of the hyperparameters of each learning algorithm, and compare multiple algorithms for classification or clustering when applied to solve the same problem.

13 Statistic Classification

13.1 Discriminative vs. Generative Methods for Classification

Many supervised learning algorithms in both regression and classification are based on some assumed learning model with a set of parameters generally represented by a vector $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_m]^T$. Once a specific model to use is determined, the main learning task is to obtain $\boldsymbol{\theta}$ based on available training dataset.

All such learning methods can be characterized based on either of the following two types of models:

- *Discriminative models*

  A discriminative algorithm is typically a deterministic method aiming at establishing a model that fits the training set in some optimal way, such as a set of decision boundaries that partitions the feature space into different regions each including all data points with the same labeling. In this case, only those data samples close to the boundaries play an important role, while all other samples farther away from the boundaries may be mostly ignored.

  Here are some properties of the Discriminative methoda:

  – The discriminative methods find the decision boundary in the feature space directly based on the data points in the training set, in general they
    - are simpler than the generative approach requiring problem-specific knowledge for building models of the data,
    - are effective in producing accurate result when the dataset is large
    - provide no insight or interpretation regarding the data and no uncertainty estimate

    Typical discriminative methods include:
    - K-nearest neighbors method
    - Logistic regressions
    - Support vector machines
    - Perceptrons and Neural networks
    - Decision trees and random forests

  – *Generative models*

    The generative methods formulate the learning problem in a probabilistic framework to establish a model for the underlying structure

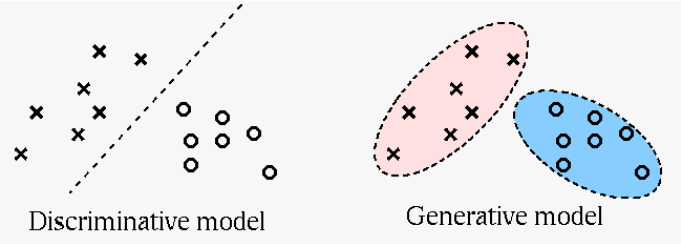


Figure 13.1 Discriminative vs. Generative Models

of the data as an effort to explain how the data was generated, and then finds specifically the decision boundaries based the model. Typically such a method finds certain probabilities of interest, such as the joint probability $p(\mathbf{x}, y|\boldsymbol{\theta})$ of both $\mathbf{x}$ and the corresponding y in the training set, or the conditional probability $p(y|\mathbf{x}, \boldsymbol{\theta})$ of the class symbol y, given the observed pattern $\mathbf{x}$. In either case, the probability is parameterized by $\boldsymbol{\theta}$, to be estimated in the learning process based on the training set. Once $\boldsymbol{\theta}$ is obtained, the probabilistic model becomes available, based on which the class identity of any unlabeled sample can be determined. For example, once the conditional probability $p(y|\mathbf{x})$ is available, we can classify a data point $\mathbf{x}$ to class labeled by y with the greatest $p(y|\mathbf{x})$, and we can further get the confidence for such a decision measured by the probability. For a probabilistic method, all data samples in the training set are needed for the estimation their overall probability densidy distribution.

Here are the properties of the generative methods:

- allow the use of problem-specific knowledge for building the model,
- can provide explanation and interpretation of the data,
- may be less prone to overfitting than a discriminative method,
- can provide uncertainty estimate
- may not be as accurate as the discriminative methods if the model does not fit the dataset well.

Typical generative methods include:

- Minimum distance classifier
- Naive Bayes classifier
- Gaussian mixture classifier
- Hidden Markov model

These two different models are illustrated in Fig. 13.1.

13.2 K Nearest Neighbors and Minimum Distance Classifiers

Here we first consider a set of simple supervised classification algorithms that assign an unlabeled sample $\mathbf{x}$ to one of the K known classes based on set N of *training samples* $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, where each sample $\mathbf{x}_n$ is labeled by $y_n = k \in \{1, \cdots, K\}$, indicating it belongs to class C_k.

- **k Nearest neighbors (k-NN) Classifier**

  Given an unlabeled pattern $\mathbf{x}$, we first find its k *nearest neighbors* in the training dataset, and then assign $\mathbf{x}$ to one of the K classes by a majority vote of the k neighbors based on their class labelings. The voting can be weighted so that closer neighbors are more heavily weighted than those that are farther away. In particular, when $k = 1$, $\mathbf{x}$ is assigned to the class of its closest neighbor.

  While the k-NN method is simple and straight forward, its computational cost is high as classifying any unlabeled pattern $\mathbf{x}$ requires computing distances to all data points in the training set.

- **Minimum Distance Classifier**

  The minimum distance classifier is based on the most straight forward idea that an unlabeled data point $\mathbf{x}$ should be classified into the closest class in the feature space, among all K such classes $\{C_1, \cdots, C_K\}$ in the training dataset:

$$\text{if } d(\mathbf{x}, C_k) = min\{d(\mathbf{x}, C_i) \;\; i = 1, \cdots, C\} \quad \text{then } \mathbf{x} \in C_k \tag{13.1}$$

  where $d(\mathbf{x}, C_k)$ can be any of the point-to-class distances considered in Section 9.1, such as the centroid distance $d(\mathbf{x}, \mathbf{m}_k) = ||\mathbf{x} - \mathbf{m}||$, and the Mahalanobis distance:

$$d_M(\mathbf{x}, C_k) = (\mathbf{x} - \mathbf{m}_k)^T \mathbf{\Sigma}_k^{-1} (\mathbf{x} - \mathbf{m}_k) \tag{13.2}$$

  based on the estimated mean and covariance matrix of all data points belonging to the kth class C_k:

$$\mathbf{m}_k = \frac{1}{N_k} \sum_{n=1}^{N_k} \mathbf{x}_n, \qquad \mathbf{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N_k} (\mathbf{x}_n - \mathbf{m})^T (\mathbf{x}_n - \mathbf{m}) \tag{13.3}$$

  We need to compare different point-to-class distances to see which of them measures the similarity between $\mathbf{x}$ and each class C_k most accurately, as shown in the following examples.

Example 13.1 Here we consider a simple binary classification problem in a 1-D space as illustrated in the left panel in Fig. 13.2, based on the centroid distance, the Euclidean distance $d_2(\mathbf{x}, \mathbf{m}_k)$ between the data point and the mean of all samples in each of the two classes represented by their

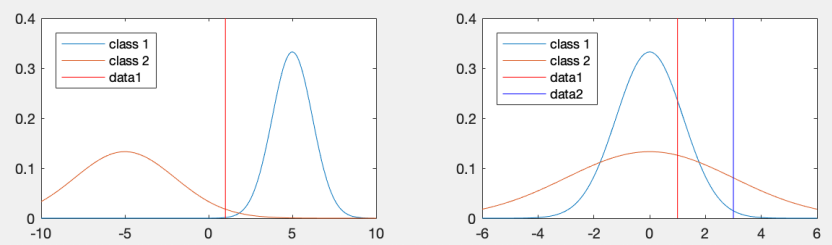


Figure 13.2 Binary Classification in 1-D

corresponding Gaussian pdfs:

$$C_1 \sim \mathcal{N}(m_1, \sigma_1^2) = \mathcal{N}(5, 1.2^2), \qquad C_2 \sim \mathcal{N}(m_2, \sigma_2^2) = \mathcal{N}(-5, 3^2) \tag{13.4}$$

Using the centroid distance based on only the two means $m_1 = 5$ and $m_2 = -5$, a data sample $x = 1$ is classified to class C_1 with a closer distance to its mean: $d_2(x, m_1) = 4 < d_2(x, m_2) = 6$. However, as shown in the figure, x seems to be more similar to class C_2, as it is more widely distribution than C_1 due to its greater variance $\sigma_2 = 3 > \sigma_1 = 1.2$, which is not taken into consideration in this case.

Example 13.2 Here we use the Mahalanobis distances between x and each of the two classes, so that the variances of the two classes as well as their means are taken into consideration. In the 1-D case this distance is simply $d_M(x, C_k) = (x - m_k)^2/\sigma_k^2$, which is proportional to $(x - m_k)^2$ but inversely proportional to σ_k^2. As $d_M(x_1, m_1) = 11.11 > d_M(x_1, m_2) = 4.0$, $x = 1$ is classified to C_2.

Example 13.3 Now consider classifying the two samples $x_1 = 1$ and $x_2 = 3$ shown in the right panel in Fig. 13.2 into either of two classes both with a normal distribution:

$$C_1 \sim \mathcal{N}(0, 1.2^2), \qquad C_2 \sim \mathcal{N}(0, 3^2) \tag{13.5}$$

The Mahalanobis distances $d_M(x_i, C_k) = (x_i - m_k)^2/\sigma_k^2$, $(i = 1, 2, k = 1, 2)$ between each of the two samples and each of the two classes are listed below:

	class 1	class 2
x_1	$1/1.2^2 = 0.69$	$1/3^2 = 0.11$
x_2	$3^3/1.2^2 = 6.25$	$3^2/3^2 = 1$

(13.6)

As $m_1 = m_2 = 0$, $|x_i - m_1|^2 = |x_i - m_2|^2$ for both x_1 and x_2, their Mahalanobis distances are determined by the variances σ_2^2, σ_2^2 alone, and they are both classified into C_2, with a greater variance $\sigma_2^2 > \sigma_2^2$ and therefore

smaller distance. However, as shown in the figure, it seems $x_1 = 1$ should be classified to C_1. We therefore see that sometimes minimum distance classification based on the Mahalanobis distance may not be reliable, and some better method need to be considered, to be discussed in the following section.

13.3 Naive Bayes Classification

The method of naive Bayes (NB) classification is a classical supervised classification algorithm, which is first trained by a training set of N samples $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ and their corresponding labelings $\mathbf{y} = [y_1, \cdots, y_N]^T$, and then used to classify any unlabeled sample $\mathbf{x}T$ into class C_k with the maximum *posterior probability*. As indicated by the name, naive Bayes classification is based on Bayes' theorem:

$$P(C_k|\mathbf{x}) = \frac{p(\mathbf{x}, C_k)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|C_k)P(C_k)}{p(\mathbf{x})} \tag{13.7}$$

where

- $P(C_k)$, or briefly P_k, is the *prior probability* that any randomly selected sample $\mathbf{x}$ belongs to class C_k without any prior knowledge of the sample. If the training set is a fair representation of all patterns of different classes in the entire population, in the sense that the data points in training set are uniformly sampled from the distribution they represent, then we can assume:

$$P_k = \frac{N_k}{N} = \frac{N_k}{\sum_{l=1}^{K} N_l}, \qquad (k = 1, \cdots, K) \qquad \text{satisfying} \qquad \sum_{k=1}^{K} P_k = 1 \tag{13.8}$$

  where N_k is the number of data samples in the training set labeled to belong to class C_k. This estimation is based on the assumption that the training samples are evenly drawn from the entire population, and are therefore a fair representation of all K classes.
- $p(\mathbf{x}|C_k) = L(C_k|\mathbf{x})$ is the *likelihood* for any observed $\mathbf{x}$ to belong to C_k, which is proportional to the conditional probability of $\mathbf{x}$ given $\mathbf{x} \in C_k$, assumed to be a Gaussian in terms of the mean vector $\mathbf{m}_k$ and covariance matrix $\mathbf{\Sigma}_k$:

$$p(\mathbf{x}|C_k) = \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \mathbf{\Sigma}_k) = \frac{1}{(2\pi)^{d/2} \left|\mathbf{\Sigma}_k\right|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \mathbf{m}_k)^T \mathbf{\Sigma}_k^{-1} (\mathbf{x} - \mathbf{m}_k)\right] \tag{13.9}$$

  This assumption is based on the fact that the Gaussian distribution has the maximum entropy (uncertainty) among all probability density functions with the same covariance, i.e., it imposes the least amount of unsupported constraint on the model for the dataset.

- $p(\mathbf{x}, C_k)$ is the joint probability of $\mathbf{x}$ and C_k:

$$p(\mathbf{x}, C_k) = p(\mathbf{x}|C_k)P(C_k) = P(C_k|\mathbf{x})p(\mathbf{x}) \tag{13.10}$$

- $p(\mathbf{x})$ is the distribution of any data sample in the dataset, independent of its class identity, the weighted sum of all likelihood $p(\mathbf{x}|C_k)$ for $k = 1, \cdots, K$, i.e., the joint probability $p(\mathbf{x}, C_k)$ marginalized over all K classes:

$$\begin{aligned} p(\mathbf{x}) &= \sum_{k=1}^{K} p(\mathbf{x}, C_k) = \sum_{k=1}^{K} P_k \, p(\mathbf{x}|C_k) \\ &= \sum_{k=1}^{K} P_k \left[\frac{1}{(2\pi)^{d/2} \left|\mathbf{\Sigma}_k\right|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{m}_k)^T \mathbf{\Sigma}_k^{-1} (\mathbf{x} - \mathbf{m}_k) \right) \right] \end{aligned} \tag{13.11}$$

- $P(C_k|\mathbf{x})$ is the posterior probability that a data sample $\mathbf{x}$ belongs to class C_k based on the observed values of $\mathbf{x}$.

Based on Bayes' theorem given above, the naive Bayes method classifies an unlabeled data sample $\mathbf{x}$ to class C_k with the maximum posterior probability (*maximum a posteriori (MAP)*):

$$\text{if } \; P(C_k|\mathbf{x}) = \max_l \{P(C_l|\mathbf{x}), \; (l = 1, \cdots, K)\}, \quad \text{then } \; \mathbf{x} \in C_k \tag{13.12}$$

As $p(\mathbf{x})$ is common to all K classes (independent of k), it plays no role in the relative comparison among the K classes, and can therefore be dropped, i.e., $\mathbf{x}$ is classified to C_k with maximal $p(\mathbf{x}|C_k)\, P_k$. In the case where the P_k is not available due to the lack of any prior knowledge, it has to be assumed to be the same for all K classes (equaly likely), then it is dropped and the method becomes maximum likelihood estimation (MLE).

The naive Bayes classifier is an optimal classifier in the sense that the classification error is minimum. To see this, consider an arbitrary boundary between two classes C_1 and C_2 that partitions the 1-D feature space into two regions R_1 and R_2, as shown in Fig. 13.3. The probability of a misclassification is the joint probability $P(\mathbf{x} \in C_i \cap \mathbf{x} \in R_j)$ for a sample $\mathbf{x} \in C_i$ but falling in R_j. The total probability of error due to misclassification can be expressed as:

$$\begin{aligned} P(error) &= P((\mathbf{x} \in R_2) \cap (\mathbf{x} \in C_1)) + P((\mathbf{x} \in R_1) \cap (\mathbf{x} \in C_2)) \\ &= P(\mathbf{x} \in R_2/C_1)\, P_1 + P(\mathbf{x} \in R_1/C_2)\, P_2 \\ &= P_1 \int_{R_2} p(\mathbf{x}/C_1) d\mathbf{x} + P_2 \int_{R_1} p(\mathbf{x}/C_2) d\mathbf{x} \end{aligned} \tag{13.13}$$

As shown in Fig. 13.3, it is obvious that the Bayes classifier is indeed optimal, due to the fact that the decision boundary $p(\mathbf{x}|C_1)P_1 = p(\mathbf{x}|C_2)P_2$ between two classes C_1 and C_2 (the bottom plot) guarantees the classification error (shaded area) to be minimum.

To find the likelihood function $p(\mathbf{x}|C_k) = \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \mathbf{\Sigma}_k)$, we need to first

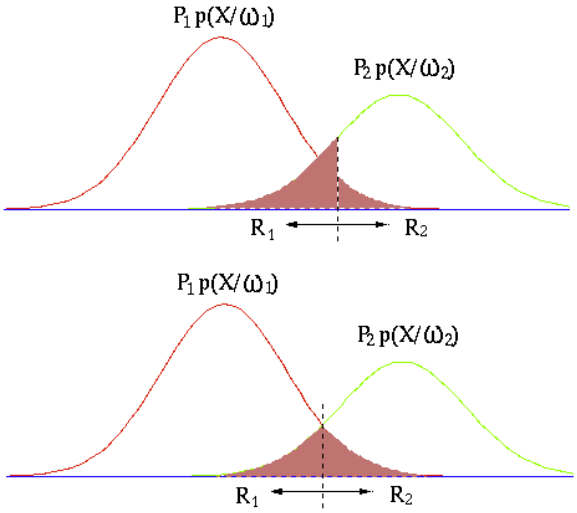


Figure 13.3 Minimum Error of Naive Bayes Classifier

estimate $\mathbf{m}_k$ and $\mathbf{\Sigma}_k$ based on the N_k training samples in class C_k in the training set:

$$\mathbf{m}_k = \frac{1}{N_k}\sum_{i=1}^{N_k}\mathbf{x}_i, \qquad \mathbf{\Sigma}_k = \frac{1}{N_k}\sum_{i=1}^{N_k}(\mathbf{x}_i - \mathbf{m}_k)(\mathbf{x}_i - \mathbf{m}_k)^T, \qquad (\mathbf{x}_i \in C_k) \quad (13.14)$$

Once both $P_k = N_k/N$ and $p(\mathbf{x}|C_k) = \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \mathbf{\Sigma}_k)$ are available, the classifier is trained, and the posterior $P(C_k|\mathbf{x})$ can be calculated for the classification in Eq. (13.12).

As the classification is based on the relative comparison of the posterior $P(C_k|\mathbf{x})$, any monotonic mapping of the posterior, such as the logarithmic function, can be equivalently used to simplify the computation:

$$\begin{aligned}\log P(C_k|\mathbf{x}) &= \log\left[p(\mathbf{x}|C_k)P_k/p(\mathbf{x})\right] = \log p(\mathbf{x}|C_k) + \log P_k - \log p(\mathbf{x}) \\ &= -\frac{1}{2}(\mathbf{x} - \mathbf{m}_k)^T\mathbf{\Sigma}_k^{-1}(\mathbf{x} - \mathbf{m}_k) \\ &\quad -\frac{N}{2}\log(2\pi) - \frac{1}{2}\log|\mathbf{\Sigma}_k| + \log P_k - \log p(\mathbf{x}) \qquad (13.15)\end{aligned}$$

We can further drop the constant terms $\log p(\mathbf{x})$ and $N\log(2\pi)/2$ shared by all classes and get the *quadratic discriminant function*:

$$D_k(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mathbf{m}_k)^T\mathbf{\Sigma}_k^{-1}(\mathbf{x} - \mathbf{m}_k) - \frac{1}{2}\log|\mathbf{\Sigma}_k| + \log P_k \qquad (13.16)$$

Now the classification can be represented by

$$\text{if } D_l(\mathbf{x}) = \max\{D_k(\mathbf{x}),\ (k = 1, \cdots, K)\}, \quad \text{then } \mathbf{x} \in C_l \qquad (13.17)$$

Equivalently, we can also treat the negative of this discriminant function as a

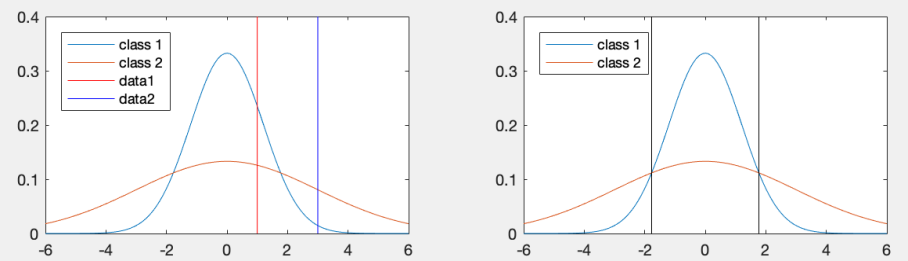


Figure 13.4 Binary Classification in 1-D Revisited

distance between point $\mathbf{x}$ and class C_k with $\mathbf{m}_k$ and $\mathbf{\Sigma}_k$:

$$d_k(\mathbf{x}, C_k) = (\mathbf{x} - \mathbf{m}_k)^T \mathbf{\Sigma}_k^{-1} (\mathbf{x} - \mathbf{m}_k) + \log |\mathbf{\Sigma}_k| - 2 \log P_k \tag{13.18}$$

for minimum distance classification. We note that the first term is the Mahalanobis distance between $\mathbf{x}$ and C_k. However, the additional terms in the expression contribute to better classification performance, as illustrated in the example below.

Example 13.4 Reconsider Example 13.3 shown in the left panel of Fig. 13.4, but now using the negative of the discriminant function treated in Eq. (13.18) as the distance. In this case, the last term $logP_k$ of the distance are dropped as they are assumed to be the same for both classes, and the distances $d(x_i, C_k) = (x_i - m_k)^2/\sigma_k^2 - \log|\sigma_k^2|,\ (i = 1, 2,\ k = 1, 2)$ between each of the two samples and each of the two classes are listed below:

	class 1	class 2
x_1	$0.69 + \log(1.2^2) = 1.06$	$0.11 + \log(3^2) = 2.31$
x_2	$6.25 + \log(1.2^2) = 6.61$	$1 + \log(3^2) = 3.20$

(13.19)

As $d(x_1, C_1) = 1.06 < d(x_1, C_2) = 2.31$, x_1 is classified into class C_1 as desired, and as $d(x_1, C_1) = 1.06 < d(x_1, C_2) = 2.31$, x_2 is still classified into C_2, the same as before. We see that the previous misclassification of x_1 into C_2 based on the Mahalanobis distance is now corrected, due to the additional term $\log(\sigma^2)$. Moreover, the right panel of Fig. 13.4 shows how the 1-D space is partitioned by the decision boundaries (the two vertical lines) into regions for C_1 (inside the two vertical lines) and C_2 (outside the vertical lines).

In the more general case of higher dimensional multiple classes, the feature space is partitioned into regions corresponding to the K classes by the decision boundaries between every pair of classes C_i and C_j, represented by the equation

$D_i(\mathbf{x}) = D_j(\mathbf{x})$:

$$\begin{aligned} &-\frac{1}{2}(\mathbf{x}-\mathbf{m}_i)^T\boldsymbol{\Sigma}_i^{-1}(\mathbf{x}-\mathbf{m}_i) - \frac{1}{2}\log|\boldsymbol{\Sigma}_i| + \log P_i \\ =\ &-\frac{1}{2}(\mathbf{x}-\mathbf{m}_j)^T\boldsymbol{\Sigma}_j^{-1}(\mathbf{x}-\mathbf{m}_j) - \frac{1}{2}\log|\boldsymbol{\Sigma}_j| + \log P_j \end{aligned} \tag{13.20}$$

which can be written as

$$\mathbf{x}^T\mathbf{W}\mathbf{x} + \mathbf{w}^T\mathbf{x} + w = 0 \tag{13.21}$$

where

$$\begin{aligned} \mathbf{W} &= -\frac{1}{2}(\boldsymbol{\Sigma}_i^{-1} - \boldsymbol{\Sigma}_j^{-1}) \\ \mathbf{w} &= \boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i - \boldsymbol{\Sigma}_j^{-1}\mathbf{m}_j \\ w &= -\frac{1}{2}(\mathbf{m}_i^T\boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i - \mathbf{m}_j^T\boldsymbol{\Sigma}_j^{-1}\mathbf{m}_j) - \frac{1}{2}\log\frac{|\boldsymbol{\Sigma}_i|}{|\boldsymbol{\Sigma}_j|} + \log\frac{P_i}{P_j} \end{aligned} \tag{13.22}$$

In general, this decision boundary is a quadratic hypersurface (hypersphere, hyperellipsoid, hyperparabola, or hyperhyperbola), by which an unlabeled data sample $\mathbf{x}$ is classified into either of the two classes C_i and C_j based on whether it is on the positive or negative side of the surface::

$$\text{if } \mathbf{x}^T\mathbf{W}\mathbf{x} + \mathbf{w}^T\mathbf{x} + w \begin{cases} > 0 \\ < 0 \end{cases}, \quad \text{then } \mathbf{x} \in \begin{cases} C_i \\ C_j \end{cases} \tag{13.23}$$

We further consider several special cases:

- All classes have the same prior:

$$P_i = P_j \quad (i, j = 1, \cdots, K) \tag{13.24}$$

  then the last term of $D_i(\mathbf{x})$ is zero, and the naive Bayes classification as a MAP method becomes an MLE method, we have

$$D_k(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mathbf{m}_k)^T\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}-\mathbf{m}_k) - \frac{1}{2}\log|\boldsymbol{\Sigma}_k| \tag{13.25}$$

- All classes have the same covariance matrix $\boldsymbol{\Sigma}_i = \boldsymbol{\Sigma}$, the discriminant function becomes:

$$D_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mathbf{m}_i)^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\mathbf{m}_i) + \log P_i \tag{13.26}$$

  Moreover, if all P_i are the same and the second term is dropped, $D_i(\mathbf{x})$ becomes the negative *Mahalanobis distance*.

  The boundary equation $D_i(\mathbf{x}) = D_j(\mathbf{x})$ between C_i and C_j becomes

$$-\frac{1}{2}(\mathbf{x}-\mathbf{m}_i)^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\mathbf{m}_i) + \log P_i = -\frac{1}{2}(\mathbf{x}-\mathbf{m}_j)^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\mathbf{m}_j) + \log P_j \tag{13.27}$$

  As the quadratic terms $\mathbf{x}^T\boldsymbol{\Sigma}^{-1}\mathbf{x}$ on both sides of the equation are the same, it becomes a linear equation:

$$\mathbf{w}^T\mathbf{x} + w = 0 \tag{13.28}$$

where

$$\mathbf{w} = \mathbf{\Sigma}^{-1}(\mathbf{m}_i - \mathbf{m}_j) \tag{13.29}$$

and

$$w = -\frac{1}{2}(\mathbf{m}_i^T \mathbf{\Sigma}^{-1} \mathbf{m}_i - \mathbf{m}_j^T \mathbf{\Sigma}^{-1} \mathbf{m}_j) + \log \frac{P_i}{P_j} \tag{13.30}$$

This linear equation represents a hyperplane between the two points $\mathbf{m}_i$ and $\mathbf{m}_j$ and perpendicular to the straight line $\mathbf{\Sigma}^{-1}(\mathbf{m}_i - \mathbf{m}_j)$ (the straight line $\mathbf{m}_i - \mathbf{m}_j$ rotated by matrix $\mathbf{\Sigma}^{-1}$).

- All classes have the same isotropic distribution:

$$\mathbf{\Sigma}_i = \sigma^2 \mathbf{I} = \text{diag}[\sigma^2, \cdots, \sigma^2] \tag{13.31}$$

then $|\mathbf{\Sigma}_i| = \sigma^{2d}$ and $D_i(\mathbf{x})$ becomes

$$D_i(\mathbf{x}) = -\frac{||\mathbf{x} - \mathbf{m}_i||^2}{2\sigma^2} + \log P_i \tag{13.32}$$

Note that the term $\log |\mathbf{\Sigma}_i|$ has been dropped from the original expression of $D_i(\mathbf{x})$ as it is now the same for all classes.

The boundary equation $D_i(\mathbf{x}) = D_j(\mathbf{x})$ becomes:

$$\frac{||\mathbf{x} - \mathbf{m}_i||^2}{2\sigma^2} - \log P_i = \frac{||\mathbf{x} - \mathbf{m}_j||^2}{2\sigma^2} - \log P_j \tag{13.33}$$

which can be simplified to a linear equation:

$$\mathbf{w}^T \mathbf{x} + w = 0 \tag{13.34}$$

where

$$\mathbf{w} = \mathbf{m}_i - \mathbf{m}_j, \quad w = -(\mathbf{m}_i^T \mathbf{m}_i - \mathbf{m}_j^T \mathbf{m}_j) + 2\sigma^2 \log \frac{P_i}{P_j} \tag{13.35}$$

This linear equation represents a hyperplane between the two points $\mathbf{m}_i$ and $\mathbf{m}_j$ and perpendicular to the straight line passing through these points.

- Further, if all classes have the same P_i, $D_i(\mathbf{x})$ becomes

$$D_i(\mathbf{x}) = -(\mathbf{x} - \mathbf{m}_i)^T(\mathbf{x} - \mathbf{m}_i) = -||\mathbf{x} - \mathbf{m}_i||^2 \tag{13.36}$$

and the Bayes classifier becomes minimum distance classifier based on the centroid distance, the Euclidean distance between $\mathbf{x}$ and the mean $\mathbf{m}_k$ of class C_k (maximizing $D_i(\mathbf{x})$ is equivalent to minimizing $||\mathbf{x} - \mathbf{m}_i||^2$).

The Matlab functions for training and testing are listed low:

```
function [M S P]=NBtraining(X,y)   % naive Bayes training
    [d N]=size(X);                 % dimensions of dataset
    K=length(unique(y));           % number of classes
    M=zeros(d,K);                  % mean vectors
    S=zeros(d,d,K);                % covariance matrices
    P=zeros(1,K);                  % prior probabilities
```

```
    for k=1:K                       % for each of K classes
        idx=find(y==k);             % indices of samples
        Xk=X(:,idx);                % get all samples in class
        P(k)=length(idx)/N;         % prior probability
        M(:,k)=mean(Xk');           % mean vector
        S(:,:,k)=cov(Xk');          % covariance matrix
    end
end

function yhat=NBtesting(X,M,S,P)    % naive Bayes testing
    [d N]=size(X);                  % dimensions of dataset
    K=length(P);                    % number of classes
    for k=1:K
        InvS(:,:,k)=inv(S(:,:,k)); % inverse of covariance
        Det(k)=det(S(:,:,k));       % determinant of covariance
    end
    for n=1:N                       % for each of N samples
        x=X(:,n);
        dmax=-inf;
        for k=1:K
            xm=x-M(:,k);
            d=-(xm'*InvS(:,:,k)*xm)/2;
            d=d-log(Det(k))/2+log(P(k)); % discriminant function
            if d>dmax
                yhat(n)=k;          % assign nth sample to kth class
                dmax=d;
            end
        end
    end
end
```

The classification result in terms of the estimated class identity $\hat{\mathbf{y}}$ of the training samples compared with the ground truth class labeling $\mathbf{y}$ can be represented in the form of a $K \times K$ confusion matrix, of which the element in the ith row and jth column is the number of training samples labeled as a member of the ith class but actually classified into the jth class. For an ideal classifier that classifies all data samples correctly, this confusion matrix should be a diagonal matrix. But for a nonideal classifier, its error rate can be found as the percentage of the number of misclassified samples (sum of all off-diagonal elements) out of the total number of samples. The code below shows the function that generates such a confusion matrix based on the estimated class identity $\hat{\mathbf{y}}$ and the ground truth labeling $\mathbf{y}$:

```
function [Cm er]=ConfusionMatrix(yhat,ytrain)
    N=length(ytrain);               % number of test samples
```

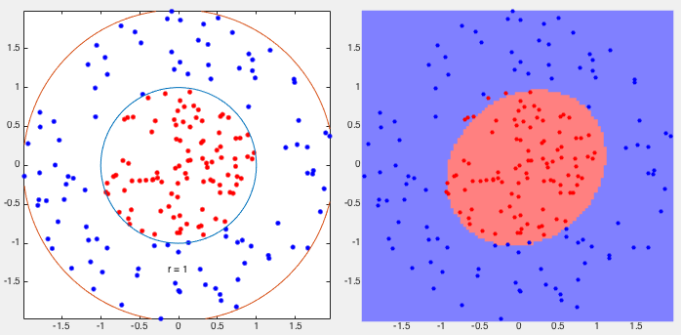


Figure 13.5 Naive Bayes Classification of Concentric Binary Classes

```
K=length(unique(ytrain));     % number of classes
Cm=zeros(K);                  % the confusion matrixxs
for n=1:N
    i=ytrain(n);
    j=yhat(n);
    Cm(i,j)=Cm(i,j)+1;
end
er=1-sum(diag(Cm))/N;         % error percentage
end
```

Example 13.5 This example shows the classification of two cocentric classes with one surrounding the other in 2-D space, as shown in Fig. 13.5. The feature space is partitioned into two regions by an ellipse and 9 out of 200 samples misclassified, i.e., with an error rate of $9/200 = 4.5\%$. The confusion matrix is shown below:

$$\begin{bmatrix} 92 & 8 \\ 1 & 99 \end{bmatrix}$$

indicating 8 of the 100 samples belonging to class 1 are misclassified to class 2, and 1 of the 100 samples belonging to class 2 is misclassified to class 1.

Example 13.6 This example shows the classification of a 2-class exclusive-or (XOR) data set with significant overlap, as shown in Fig. 13.6, in terms of the confusion matrix and the error rate of $61/400 = 15.25\%$. Note that the decision boundaries are a pair of hyperbolas. The confusion matrix is

$$\begin{bmatrix} 175 & 25 \\ 36 & 164 \end{bmatrix}$$

Example 13.7 The left panel of Fig. 13.7 shows data points in 2-D space drawn from three distributions shown in the middle panel, while the right one shows the partitioning of the space by quadratic decision boundaries obtained by the

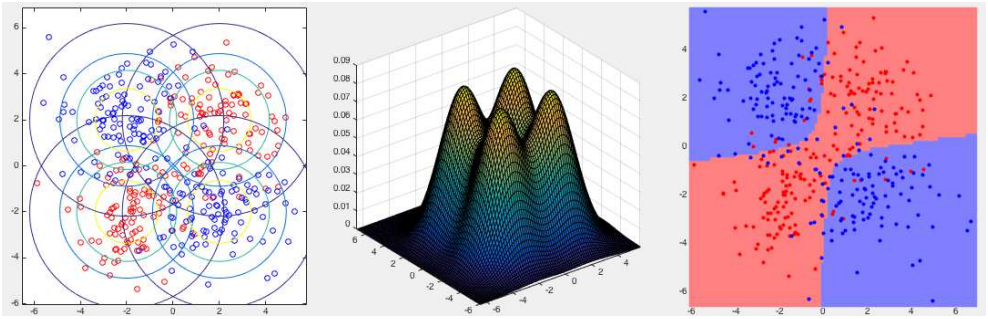


Figure 13.6 Naive Bayes Classification of XOR Binary Classes

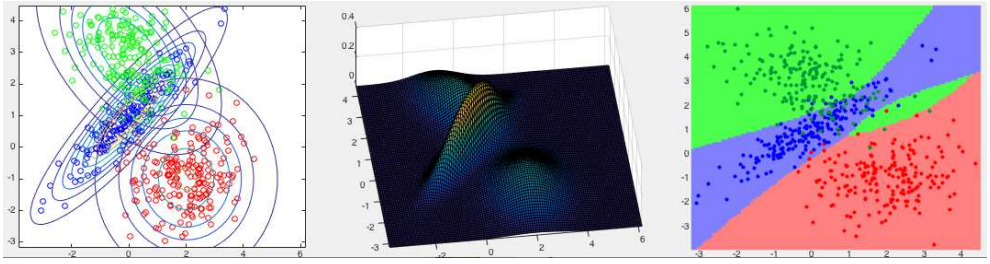


Figure 13.7 Naive Bayes Classification of Three Classes

naive Bayes classifier. The confusion matrix of the classification result is shown below, with the error rate $49/600 = 8.17\%$.

$$\begin{bmatrix} 196 & 3 & 1 \\ 0 & 191 & 9 \\ 3 & 33 & 164 \end{bmatrix}$$

Example 13.8 Fig. 13.8 shows a 3-D feature space containing $K = 3$ groups of data points (in red, green and blue), which is partitioned by three quadratic decision boundaries $D_i(\mathbf{x}) = D_j(\mathbf{x})$ (Eq. (13.21)) into three regions for each C_l of the three classes with maximum discriminant function $D_l(\mathbf{x})$ (Eq. (13.17).

The same classification result can also be reached by carrying out three binary classifications each based on one of the three decision boundaries followed by a majority vote. For example, a data point in the red group will win at least two out of the three binary classifications and it is therefore classified into the red class. This method can be used to carry out multiclass classification based on any binary classifier, such as the support vector machine, to be considered in Chapter 14.

Example 13.9 Fig. 13.9 shows a subset of the dataset of handwritten digits each in a 16×16 image form treated as a 256 dimensional vector.

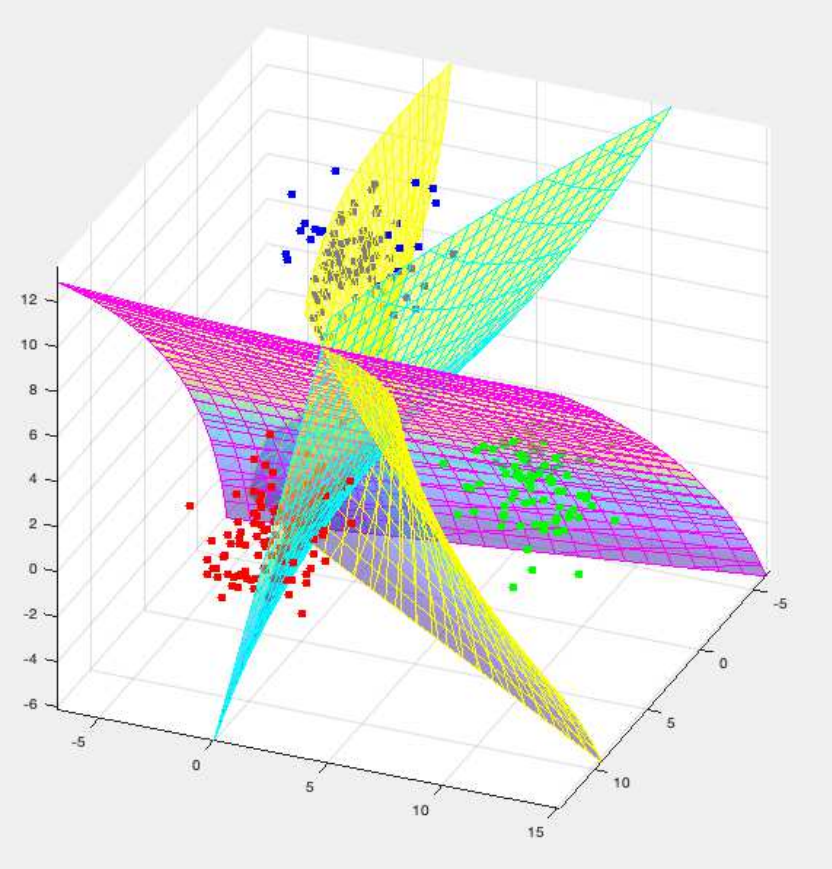


Figure 13.8 Partitioning of a 3-D Space by Quadratic Decision Boundaries

The dataset is visualized by mapping the data points from the original 256-D space to 3-D or even 2-D space based on either the linear KLT transform (Section 10.2), or the nonlinear tSNE (Section 11.6), as shown respectively in the left and right panels of Figs. 13.10 and 13.11.

To use the naive-Bayes classifier for the classification of this dataset, we need to estimate the covariance matrix $\mathbf{\Sigma}_k$ based on the 224 samples in each of the $K = 10$ class. As the rank of the matrix is $224-1 = 223 < 256$, it is not invertible and $\mathbf{\Sigma}_k^{-1}$ needed in the algorithm does not exist. We therefore have to reduce the dimensionality by, either KLT or tSNE, from 256 to some value, e.g., 100, smaller than the number of samples used for estimating the covariance matrix.

The classifier is first trained on half of the data samples randomly selected samples and then tested on the other half. The classification results are given below in terms of the confusion matrices of the taring samples (first) and the testing samples (second). All 1120 training samples are correctly classified with

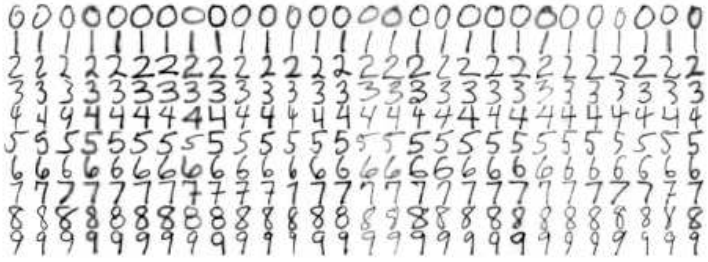


Figure 13.9 Samples of Handwritten Digits

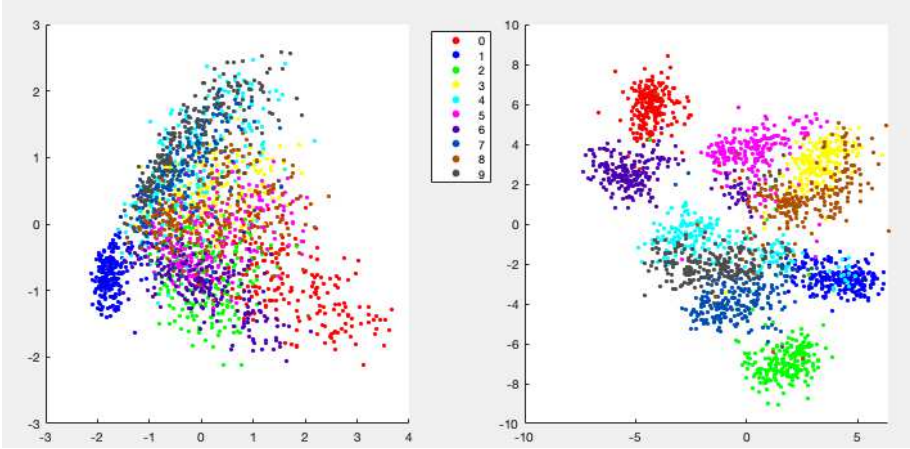


Figure 13.10 2-D Visualization based on PCA (left) and tSNE (right)

zero error rate, while out of the remaining 1120 test samples, 198 are misclassified with an error rate of 17.7%.

$$\begin{bmatrix} 107 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 108 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 105 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 118 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 110 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 120 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 128 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 104 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 109 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 111 \end{bmatrix}$$

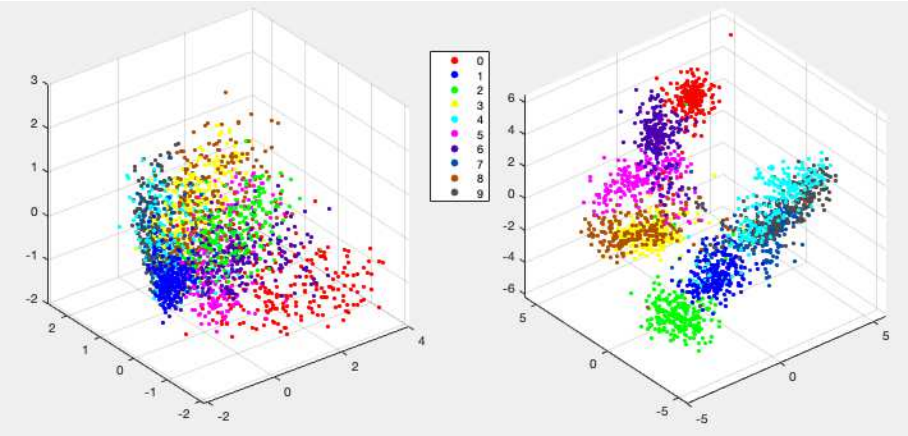


Figure 13.11 3-D Visualization based on PCA (left) and tSNE (right)

$$\begin{bmatrix} 101 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 58 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 105 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 2 & 6 & 101 & 2 & 4 & 0 & 28 & 4 & 4 \\ 0 & 15 & 0 & 0 & 102 & 0 & 0 & 4 & 0 & 9 \\ 5 & 4 & 1 & 1 & 0 & 99 & 1 & 1 & 2 & 1 \\ 0 & 17 & 1 & 0 & 2 & 0 & 95 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 62 & 0 & 0 \\ 6 & 15 & 6 & 4 & 6 & 1 & 0 & 18 & 109 & 9 \\ 0 & 3 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 90 \end{bmatrix}$$

The dimensionality of the dataset can be reduced further down to some value much lower such as a single digits, by either the PCA or tSNE methods, and then the naive Bayes classification is carried out in the low dimensional space. The error rates of the two classification methods when applied to this dataset (with all 2240 samples used in both training and testing) are compared in the table below and plotted in Fig. 13.12. It is interesting to note that when the dimensionality is reduced to lower than 7, the error rates based on tSNE are much lower than those based on PCA, but for higher dimensionalities, PCA slightly outperforms tSNE.

Dimensions:	1	2	3	4	5	6	7	8	9	10
PCA:	0.69	0.48	0.25	0.18	0.15	0.11	0.08	0.06	0.06	0.05
tSNE:	0.50	0.15	0.10	0.08	0.08	0.08	0.08	0.08	0.08	0.07

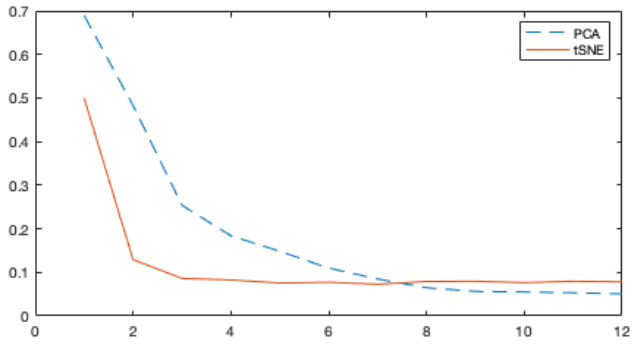


Figure 13.12 Comparison of KLT and tSNE for Feature Selection

13.4 Adaptive Boosting

Adaptive boosting (AdaBoost) is a supervised classification algorithm that belongs to the general method of ensemble learning which is first discussed in Section 4.4. AdaBoost is a binary classifier trained based on the training set containing N labeled data samples $\{(\mathbf{x}_n, y_n) \mid n = 1, \cdots, N\}$, where each sample $\mathbf{x}_n$ is labeled by $y_n \in \{-1, +1\}$, indicating to which of the two classes C_- and C_+ it belongs.

AdaBoost learning is an iterative process. In the t-th iteration, each of the N training samples is classified into one of the two classes by a *weak classifier* $h_t(\mathbf{x}_n) \in \{-1, +1\}$, as a hypothesis. Here the classifier is said to be weak in the sense that its performance only needs to be better than chance. If $\mathbf{x}_n$ is correctly classified, then $h_t(\mathbf{x}_n) = y_n = \pm 1$, i.e., $y_n\, h_t(\mathbf{x}_n) = 1$, otherwise it is misclassified with $h_t(\mathbf{x}_n) = -y_n = \pm 1$, i.e., $y_n\, h_t(\mathbf{x}_n) = -1$.

The weighted overall error rate is defined as:

$$\varepsilon_t = \frac{\sum_{y_n\, h_t(\mathbf{x}_n)=-1} w_t(n)}{\sum_{n=1}^{N} w_t(n)} \tag{13.37}$$

where $w_t(n)$ is the weight for the nth training sample $\mathbf{x}_n$ at the t-th iteration. For this error rate to be lower than chance, it is required that $\varepsilon_t < 1/2$. We need to find the best weak classifier that minimizes this error ε_t. We also find the correct rate as

$$1 - \varepsilon_t = 1 - \frac{\sum_{y_n\, h_t(\mathbf{x}_n)=-1} w_t(n)}{\sum_{n=1}^{N} w_t(n)} = \frac{\sum_{y_n\, h_t(\mathbf{x}_n)=1} w_t(n)}{\sum_{n=1}^{N} w_t(n)} \tag{13.38}$$

At the beginning of the iteration with $t = 0$, all N weights are initialized to 1, $w_0(1) = \cdots = w_0(N) = 1$, and the weighted error

$$\varepsilon_0 = \frac{1}{N} \sum_{y_n\, h_0(\mathbf{x}_n)=-1} w_0(n) = \frac{\text{number of misclassified samples}}{\text{total number of samples}} \tag{13.39}$$

is the probability for any training sample $\mathbf{x}$ to be misclassified by the weak classifier h_0. During the iteration when $t > 0$, this weight $w_t(n)$ will be modified depending on whether $\mathbf{x}_n$ is classified correctly in the subsequent iterations.

At each iteration, we also construct a *strong* or *boosted classifier* $H_t(\mathbf{x}_n)$ as a linear combination (boosting) of all previous weak classifiers $h_i(\mathbf{x}_n)$, $(i \leq t)$:

$$\begin{aligned} F_t(\mathbf{x}_n) &= \alpha_t h_t(\mathbf{x}_n) + F_{t-1}(\mathbf{x}_n) = \alpha_t h_t(\mathbf{x}_n) + \alpha_{t-1} h_{t-1}(\mathbf{x}_n) + F_{t-2}(\mathbf{x}_n) \\ &= \cdots = \sum_{i=1}^{t} \alpha_i h_i(\mathbf{x}_n) \qquad (n = 1, \cdots, N) \end{aligned} \tag{13.40}$$

where α_i is some coefficient for $h_i(\mathbf{x}_n)$ in the i-th iteration, to be determined as shown below. The strong classifier is then defined as the sign function of $F_t(\mathbf{x})$:

$$H_t(\mathbf{x}_n) = sign[F_t(\mathbf{x}_n)] = \begin{cases} +1 & F_t(\mathbf{x}_n) > 0 \\ -1 & F_t(\mathbf{x}_n) < 0 \end{cases} \tag{13.41}$$

by which a training sample $\mathbf{x}_n$ is classified into one of the two classes. The magnitude $|F_t(\mathbf{x}_n)|$ represents the confidence of this decision.

The weight $w_t(n)$ of $\mathbf{x}_n$ in the t-th iteration $(t > 0)$ is defined as

$$w_t(n) = e^{-y_n F_{t-1}(\mathbf{x}_n)} \begin{cases} > 1 & \text{if } y_n\, F_{t-1}(\mathbf{x}_n) < 0 \text{ (misclassification)} \\ < 1 & \text{if } y_n\, F_{t-1}(\mathbf{x}_n) > 0 \text{ (correct classification)} \end{cases} \tag{13.42}$$

which can be obtained from $w_{t-1}(n)$ of the previous iteration:

$$\begin{aligned} w_t(n) &= e^{-y_n\, F_{t-1}(\mathbf{x}_n)} = e^{-y_n\, [\alpha_{t-1} h_{t-1}(\mathbf{x}_n) + F_{t-2}(\mathbf{x}_n)]} \\ &= e^{-\alpha_{t-1}\, y_n h_{t-1}(\mathbf{x}_n)}\; e^{-y_n\, F_{t-2}(\mathbf{x}_n)} = e^{-\alpha_{t-1}\, y_n h_{t-1}(\mathbf{x}_n)}\; w_{t-1}(n) \end{aligned} \tag{13.43}$$

and this can be carried out recursively all the way back to the first iteration with $w_1(k) = e^{-\alpha_0\, y_n h_0(\mathbf{x}_n)} w_0(n)$.

The performance of the strong classifier can be measured by the *exponential loss*, defined as the sum of all N weights:

$$E_{t+1} = \sum_{n=1}^{N} w_{t+1}(n) = \sum_{n=1}^{N} e^{-y_n\, F_t(\mathbf{x}_n)} = \sum_{n=1}^{N} e^{-\alpha_t\, y_n\, h_t(\mathbf{x}_n)}\; w_t(n) \qquad (t > 1) \tag{13.44}$$

Now we can find coefficient α_t in Eq. (13.40) as the one that minimizes the exponential loss E_{t+1}. To do so, we first separate each of the N terms in the summation above into two parts, corresponding to whether $\mathbf{x}_n$ is classified by h_t correctly or incorrectly:

$$E_{t+1} = \sum_{n=1}^{N} w_t(n) e^{-\alpha_t\, y_n\, h_t(\mathbf{x}_n)} = \sum_{y_n h_t(\mathbf{x}_n) = -1} w_t(n) e^{\alpha_t} + \sum_{y_n h_t(\mathbf{x}_n) = 1} w_t(n) e^{-\alpha_t} \tag{13.45}$$

and then set its derivative with respect to α_t to zero:

$$\frac{dE_{t+1}}{d\alpha_t} = \sum_{y_n h_t(\mathbf{x}_n) = -1} w_t(n) e^{\alpha_t} - \sum_{y_n h_t(\mathbf{x}_n) = 1} w_t(n) e^{-\alpha_t} = 0 \tag{13.46}$$

Solving this equation we get α_t as the optimal coefficient that minimizes E_{t+1}:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{\sum_{y_n h_t(\mathbf{x}_n)=1} w_t(n)}{\sum_{y_n h_t(\mathbf{x}_n)=-1} w_t(n)} \right) = \ln \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}} > 0 \tag{13.47}$$

which is greater than zero because $\varepsilon_t < 1/2$, $1 - \varepsilon_t > 1/2$, and

$$e^{\alpha_t} = \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}} > 1, \quad e^{-\alpha_t} = \sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}} < 1 \tag{13.48}$$

We see that if a weak classifier h_t has a small error ε_t, it will be weighted by a large α_t and therefore contributes more to the strong classifier H_t; on the other hand, if h_t has a large error ε_t, it will be weighted by a small α_t and contributes less to H_t. As the strong classifier H_t takes advantage of all previous weak classifiers $h_1, \cdots, h_t$, each of which may be more effective in a certain region of the N-D feature space than others, H_t can be expected to be a much more accurate classifier than any of the weak classifiers.

Replacing t by $t+1$ in Eq. (13.43), we now get

$$w_{t+1}(n) = w_t(n)\, e^{-\alpha_t\, y_n h_t(\mathbf{x}_n)} = \begin{cases} w_t(n)\sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}} < w_t(n) & \text{if } y_n h_t(\mathbf{x}_n) = 1 \\ w_t(n)\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}} > w_t(n) & \text{if } y_n h_t(\mathbf{x}_n) = -1 \end{cases} \tag{13.49}$$

We see that if $y_n h_t(\mathbf{x}_n) = 1$, i.e., $\mathbf{x}_n$ is classified correctly by h_t, it will be weighted more lightly by $w_{t+1}(n) < w_t(n)$ in the next iteration, but if $y_n h_t(\mathbf{x}_n) = -1$, i.e., $\mathbf{x}_n$ is classified incorrectly by h_t, it will be weighted more heavily by $w_{t+1}(k) > w_t(k)$ in the next iteration, thereby it will be emphasized and have a better chance to be corrected to have more accurately classified by h_{t+1} in the next iteration.

We further consider the ratio of the exponential losses of two consecutive iterations:

$$\begin{aligned} \frac{E_{t+1}}{E_t} &= \frac{\sum_{n=1}^N w_{t+1}(n)}{\sum_{n=1}^N w_t(n)} = \frac{\sum_{n=1}^N w_t(n) e^{-\alpha_t\, y_n\, h_t(\mathbf{x}_n)}}{\sum_{n=1}^N w_t(k)} \\ &= \frac{\sum_{y_n h_t(\mathbf{x}_n)=-1} w_t(n)}{\sum_{n=1}^N w_t(n)} e^{\alpha_t} + \frac{\sum_{y_n h_t(\mathbf{x}_n)=1} w_t(n)}{\sum_{n=1}^N w_t(n)} e^{-\alpha_t} \\ &= \varepsilon_t \sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}} + (1-\varepsilon_t) \sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}} = 2\sqrt{\varepsilon_t(1-\varepsilon_t)} \le 1 \end{aligned} \tag{13.50}$$

This ratio is twice the geometric average $\sqrt{\varepsilon_t(1-\varepsilon_t)} \le 1$ of ε_t and $1-\varepsilon_t$, which reaches its maximum when $\varepsilon_t = 1 - \varepsilon_t = 1/2$. However, as $\varepsilon_t < 1/2$, the ratio is always smaller than 1. We see that the exponential cost can be approximated as an exponentially decaying function from its initial value $E_0 = \sum_{n=1}^N w_0(n) = N$ for some $\varepsilon < 1/2$:

$$\lim_{t\to\infty} E_t \approx \lim_{t\to\infty} \left(2\sqrt{\varepsilon(1-\varepsilon)}\right)^t E_0 = 0 \tag{13.51}$$

We can therefore conclude that the exponential loss will always decrease, i.e., the error of AdaBoost will always converge to zero.

The weak classifier used in each iteration is typically implemented as a *decision stump*, a binary classifier that partitions the N-D feature space into two regions. Specifically, as a simple example, a coordinate descent method can be used by which all training samples are projected onto the ith dimension of the feature space $\mathbf{d}_i$:

$$x_n = P_{\mathbf{d}_i}(\mathbf{x}_n) = \frac{\mathbf{x}_n^T \mathbf{d}_i}{||\mathbf{d}_i||}, \qquad (n = 1, \cdots, N) \tag{13.52}$$

and then classified into two classes by a threshold T along the direction of $\mathbf{d}_i$:

$$h(x_n) = \begin{cases} +1 & \text{if } x_n < T \\ -1 & \text{if } x_n > T \end{cases} \tag{13.53}$$

The optimal decision stump is obtained when the following weighted error is minimized with respect to the threshold T along that direction of $\mathbf{d}_i$:

$$\varepsilon_t = \sum_{n=1}^{N} w_t(n)\, \delta(h_t(x_n) - y_n) = \sum_{h_t(x_n) \neq y_n} w_t(n) \tag{13.54}$$

Alternatively, the weak classifier can also be obtained based on the *principal component analysis (PCA)* by partitioning the N-D feature space along the directions of the eigenvectors of the between-class scatter matrix

$$\mathbf{S}_b = \frac{1}{N}\left[N_-(\mathbf{m}_{-1} - \mathbf{m})(\mathbf{m}_{-1} - \mathbf{m})^T + N_+(\mathbf{m}_{+1} - \mathbf{m})(\mathbf{m}_{+1} - \mathbf{m})^T\right] \tag{13.55}$$

where N_- and N_+ are the numbers of samples in the two classes ($N_- + N_+ = N$), and

$$\mathbf{m}_\pm = \frac{1}{N_\pm} \sum_{y_n = \pm 1} w(n)\, \mathbf{x}_n, \qquad \mathbf{m} = \frac{1}{N} \sum_{n=1}^{N} w(n)\, \mathbf{x}_n \tag{13.56}$$

are the weighted mean vectors of the two classes and the total weighted mean vector of all N samples in the training set. The training samples are likely to be better separated along these directions of the eigenvectors of $\mathbf{S}_b$, which measures the separability of the two classes. The eigenequations of $\mathbf{S}_b$ is:

$$\mathbf{S}_b = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T \tag{13.57}$$

As $\mathbf{S}_b$ is symmetric, the eigenvector matrix $\mathbf{V}$ is orthonormal and its columns can be used as an orthogonal basis that spans the feature space as well as the standard basis. The same binary threshold classification considered above for the weak classifiers can be readily applied along these PCA bases.

The Matlab code of the main iteration loop of the algorithm is listed below, followed by the essential functions called by the main loop. Here `T` is the maximum number of iteration, and `N` is the total number of training samples.

The algorithm can be carried out in the feature space based on either the

PCA basis (columns of the eigenvector matrix $\mathbf{V}$), or the original standard basis (columns of the identity matrix $\mathbf{I}$).

The *decision stump* is implemented by a binary classifier, which partitions all training samples projected onto a 1-D space into two groups by a threshold value, which needs to be optimal in terms of the number of misclassification. A parameter `plt` is used to indicate the polarity of the binary classification, i.e., which of the two class C_1 and C_0 is on the lower (or higher) side of the threshold.

```
Plt=[];              % polarity of binary classification
Tr=[];               % transformation of basis vectors
Th=[];               % threshold of binary classification
Alpha=[];            % alpha values
h=zeros(T,N);        % h functions
F=zeros(T,N);        % F functions
w=ones(T,N);         % weights for N training samples
Er=N;                % error initialized to N
t=0;                 % iteration index
while t<T  & Er>0    % the main iteration
    t=t+1;
    [tr th plt er]=WeakClassifier(X,y,w(t,:),PCA);
                                % weak classifier
    alpha=log(sqrt((1-er)/er)); % update alpha
    Alpha=cat(1,Alpha,alpha);   % record alpha
    Tr=cat(2,Tr,tr');           % record transform vector
    Th=cat(1,Th,th);            % record threshold
    Plt=cat(1,Plt,plt);         % record polarity
    x=tr*X;                     % carry out transform
    c=sqrt(er/(1-er));
    for n=1:N                   % update weights
        h(t,n)=h_function(x(n),th,plt);  % find h function
       if h(t,n)*y(n)<0
           w(t+1,n)=w(t,n)/c;   % update weights
       else
           w(t+1,n)=w(t,n)*c;
       end
    end
    F(t,:)=Alpha(t)*h(t,:);     % get F functions
    if t>1
        F(t,:)=F(t,:)+F(t-1,:);
    end
    Er=sum(sign(F(t,:).*y)==-1);  % error of strong classifier,
                                  % number of misclassifications
    fprintf('%d: %d/%d=%f\n',t,Er,N,Er/N)
end
```

Here are the functions called by the main iteration loop above:

```
function [Tr,Th Plt Er]=WeakClassifier(X,y,w,pca)
    % X: N columns each for one of the N training samples
    % y: labeling of X
    % w: weights for N training samples
    % pca: use PCA dimension if pca~=0
    % Er: minimum error among all D dimensions
    % Tr: transform vector (standard or PCA basis)
    % Th: threshold value
    % Plt: polarity
    [D N]=size(X);
    n0=sum(y>0);            % number of samples in class C+
    n1=sum(y<0);            % number of samples in class C-
    if pca                  % find PCA basis
        for i=1:D
            Y(i,:)=w.*X(i,:);
        end
        X0=Y(:,find(y>0));  % all samples in class C+
        X1=Y(:,find(y<0));  % all samples in class C-
        m0=mean(X0')';      % mean of C+
        m1=mean(X1')';      % mean of C-
        mu=(n0*m0+n1*m1)/N; % over all mean
        Sb=(n0*(m0-mu)*(m0-mu)'+n1*(m1-mu)*(m1-mu)')/N;
                            % between-class scatter matrix
        [v d]=eig(Sb);      % eigenvector and eigenvalue of Sb
    else
        v=eye(N);           % standard basis
    end
    Er=9e9;                 % initialize min error
    for i=1:D               % for each of D dimensions
        tr=v(:,i)';         % get an eigenvector of Sb
        x=tr*X;             % rotate the vector
        [th plt er]=BinaryClassifier(x,y,w);
                            % binary classify N samples in 1-D
        er=0;
        for n=1:N
            h(n)=h_function(x(n),th,plt);
                            % h-function of nth sample
            if h(n)*y(n)<0  % if misclassified
                er=er+w(n); % add error
            end
        end
        er=er/sum(w);       % total error of ith dimension
```

```
        if er<Er            % a smaller error
            Er=er;          % update min error
            Plt=plt;        % polarity
            Th=th;          % threshold
            Tr=tr;          % transform vector
        end
    end
end

function h=h_function(x, th, plt)
    if xor(x>th, plt)
        h=1;
    else
        h=-1;
    end
end

function [Th Plt Er]=BinaryClassifier(x,y,w)
    N=length(x);
    [x1 i]=sort(x);         % sort 1-D data x
    y1=y(i);                % reorder targets
    w1=w(i);                % reorder weights
    Er=9e9;                 & initialize min error
    for n=1:N-1             % for N-1 ways of binary classification
        e0=sum(w1(find(y1(1:n)==1)))+sum(w1(n+find(y1(n+1:N)==-1)));
        e1=sum(w1(find(y1(1:n)~=1)))+sum(w1(n+find(y1(n+1:N)~=-1)));
        if e1 > e0          % polarity: left -1, right +1
            plt=0; er=e0;
        else                % polarity: left +1, right -1
            plt=1; er=e1;
        end
        if er<Er            % a smaller error
            Er=er;          % update min error
            Plt=plt;        % polarity
            Th=(x1(k)+x1(k+1))/2;  % threshold
        end
    end
end
```

Example 13.10 This example shows the classification of the two-class XOR dataset (used previously for testing the naive Bayes method), containing two classes of $N = 200$ samples each in 2-D space, represented by the red and blue dots in Fig. 13.13. The intermediate results after 4, 8, 16 and 32 iterations shows the progress of the iteration, when the error rate continuously reduces from

134/400, 85/400, 81/400, 58/100. Fig. 13.14 shows a sequence of 350 binary participations of the 2-D space (top left) and the final classification result (top right). The error rate eventually reduced to zero as guaranteed by the AdaBoost method. Although all training samples are correctly classified eventually, the result suffers the problem of overfitting.

Example 13.11 Fig. 13.15 shows the partitioning of the 2-D space by the AdaBoost algorithm trained on data samples from two concentric classes, along both the standard basis vectors (left) and the PCA directions (right). The PCA method performances significantly better in this example as its error reduces faster and it converges to zero after 61 iterations, while the error of the method based on the standard axes (in vertical and horizontal directions) does not go down to zero even after 120 iterations. This faster reduction of error of the PCA method can be easily understood as many different directions are used to partition the space into two regions, while in the coordinate descent method the partitioning of the space is limited to either of the two directions.

Fig. 13.16 shows how the AdaBoost algorithm is iteratively trained. The error rate of both the training and testing samples, are plotted as the iteration progresses (top). Also, the weighted error ε_t, the exponential costs E_t, and the ratio $E_{t+1}/E_t = 2\sqrt{\varepsilon_t(1-\varepsilon_t)}$ are all plotted (bottom). We see that $\varepsilon_t < 0.5$ and $E_{t+1}/E_t < 1$ for all steps, and the exponential cost E_t attenuates exponentially towards zero.

Example 13.12 This example shows the classification of a dataset containing 200 samples of two classes (100 each) in an XOR pattern in the 2-D space, by the AdaBoost method based on both coordinate descent and PCA methods, as shown in Figs. 13.17 and 13.18, respectively. In both cases, half of the dataset (top-left) is used for training, while the other half for testing (top-right).

We see that the coordinate descent method performs poorly in both the slow convergence (more than 400 iterations) during training and high classification error rate (more than 1/3) during testing. The partitioning of the 2-D space is an obvious over fitting of the training set instead of reflecting the actual distribution of the two classes (four clusters). On the other hand, the PCA method converges quickly (10 iterations) and classifies the testing samples with very low error rate. The space is clearly partitioned into two regions corresponding to the two classes.

Problems

Carry out classification using both the iris and the handwritten digit datasets by each of the following three methods. In each case, cross validate your algorithm by using 50% randomly chosen samples in the dataset for training and the other 50% for testing. Show your classification results in the confusion matrix together

with the error rate, the sum of all off-diagonal components of the confusion matrix divided by the total number of samples.

1. The K-nearest neighbor method
2. The minimum distance method based on Mahananobis distance
3. The naive Bayes method

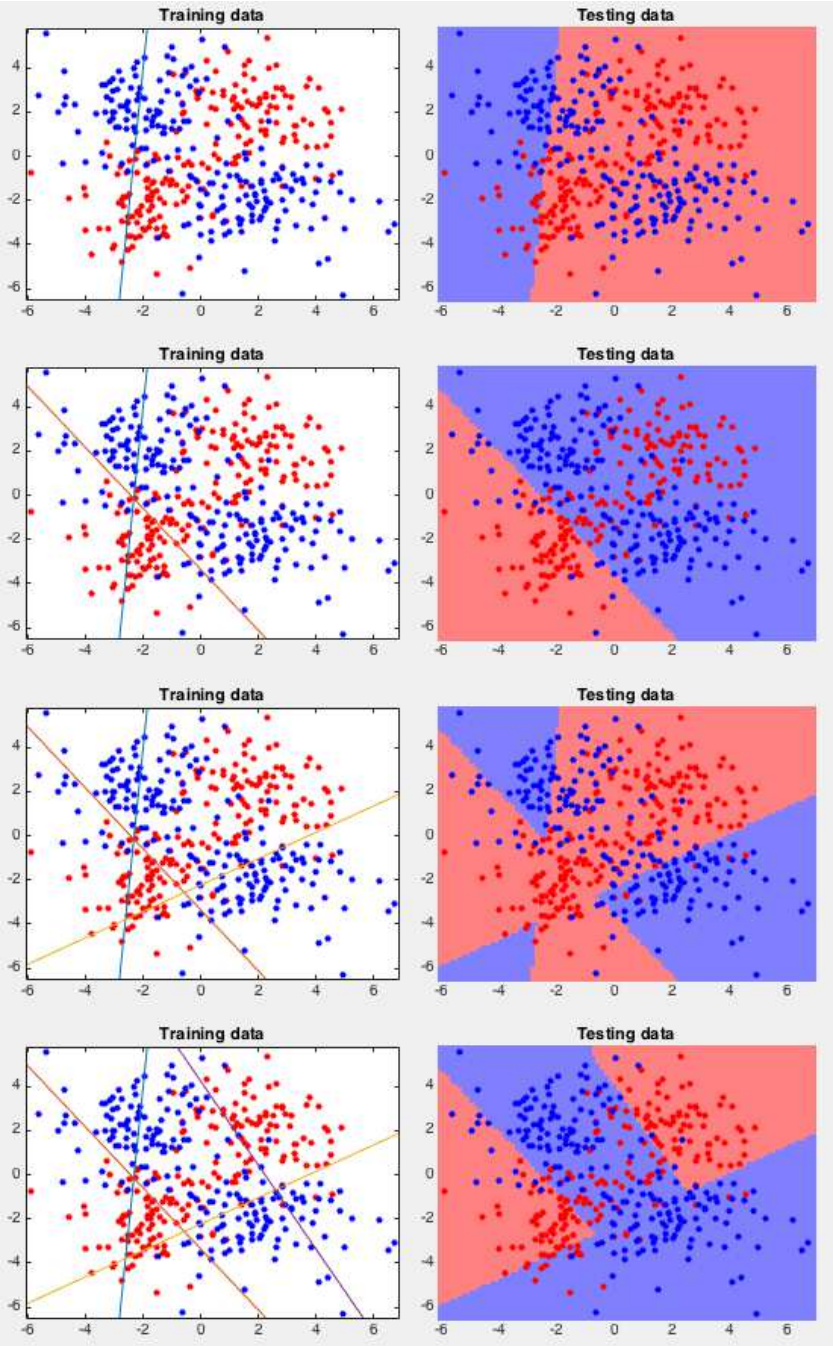


Figure 13.13 Classification by AdaBoost on XOR Dataset (early iterations)

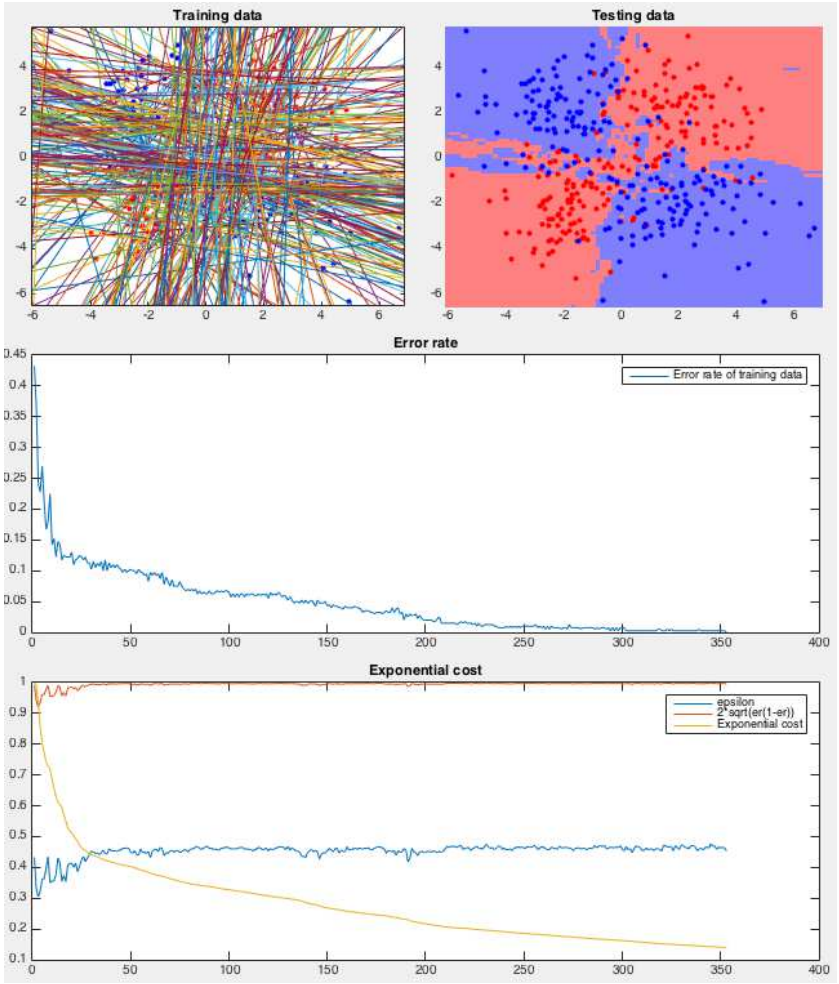


Figure 13.14 Classification by AdaBoost on XOR Dataset (final result)

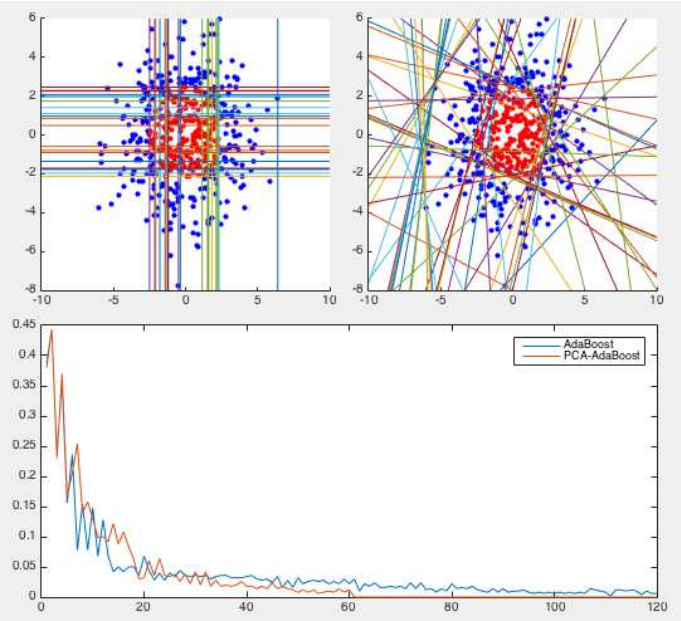


Figure 13.15 AdaBoost Classification Along Basis Vectors and PCA Directions

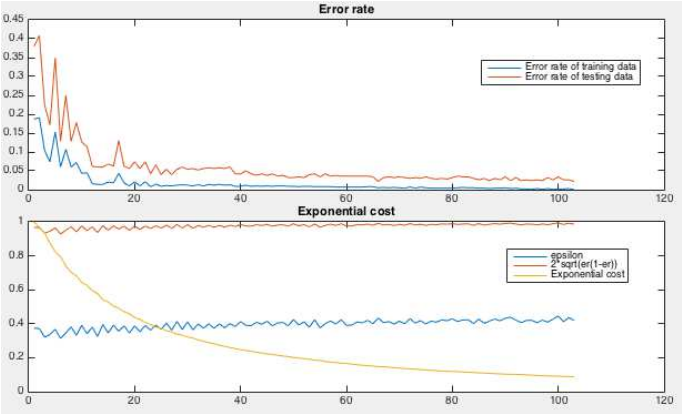


Figure 13.16 Progress of Iterative Training of AdaBoost

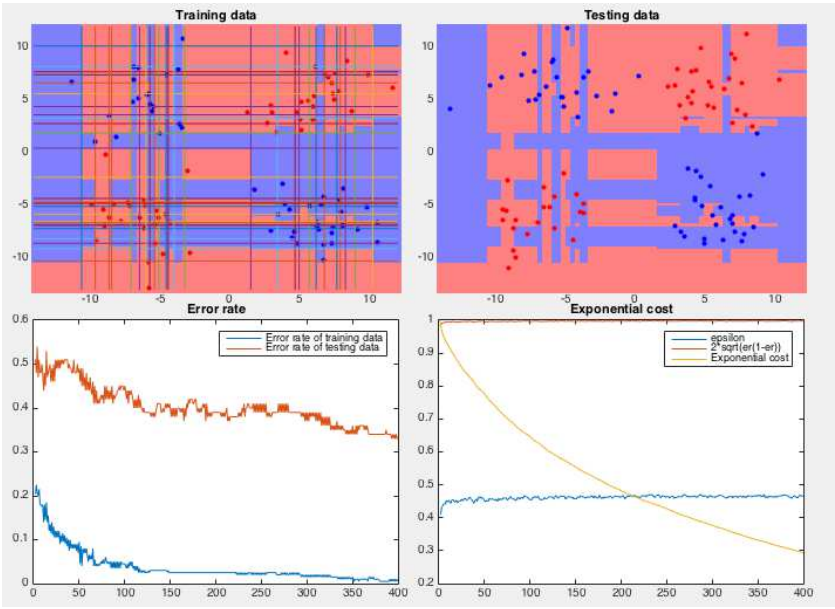


Figure 13.17 Classification of XOR Dataset (Coordinate Descent)

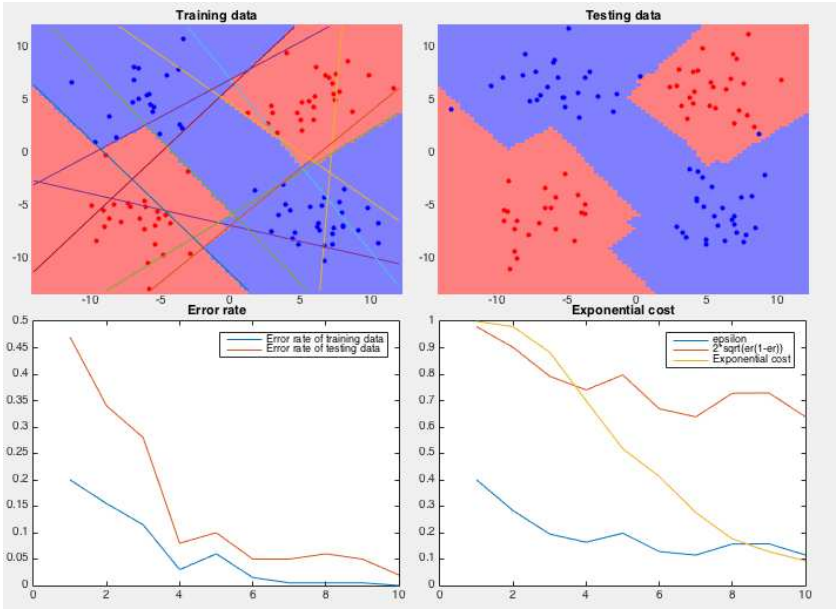


Figure 13.18 Classification of XOR Dataset (PCA method)

14 Support Vector machine

14.1 Maximum Margin and Support Vectors

The support vector machine (SVM) is a supervised binary classifier trained by a data set $\{(\mathbf{x}_n, y_n),\ \ n = 1, \cdots, N\}$, of which sample $\mathbf{x}_n$ belongs to either class C_+ if labeled by $y_n = 1$ or class C_- if labeled by $y_n = -1$. The goal of the training process is to partition the feature space into two halves by a *decision plane* H_0, described by the following linear *decision equation*:

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = \sum_{i=1}^{d} x_i w_i + b = 0 \tag{14.1}$$

in terms of its normal vector $\mathbf{w}$ and intercept b. Once these two parameters are determined in the training process, any unlabeled sample $\mathbf{x}$ can be classified into either of the two classes:

$$\text{if } \ f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b \begin{cases} > 0 \\ = 0 \\ < 0 \end{cases}, \quad \text{then } \begin{cases} \mathbf{x} \in C_+ \\ \mathbf{x} \in H_0 \\ \mathbf{x} \in C_- \end{cases} \tag{14.2}$$

Obviously the underlying assumption of this simple idea is the linear separability of the two classes. However, as we will see later, the SVM can also be applied to nonlinear and multiclass classifications.

We note that Eq. (14.1) is actually the same as Eq. (5.1) for linear regression, when the linear regression function $f(\mathbf{x})$ is thresholded by the zero plane to become an equation $f(\mathbf{x}) = 0$. This is illustrated in Fig. 14.1, which is essentially the same as Fig. 5.1. Although both SVM and linear regression methods need to obtain the optimal parameters $\mathbf{w}$ and b, they are very different in that the SVM, as a classification method, needs to determine $\mathbf{w}$ and b so that the resulting decision plane shall separate the training samples optimally, in the sense that its distance to the closest sample on either side, called *support vectors*, are maximized.

By moving the constant term in Eq. (14.1) to the right hand side and dividing both sides by $||\mathbf{x}||$, this equation can be rewritten as:

$$p_{\mathbf{w}}(\mathbf{x}) = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||} = -\frac{b}{||\mathbf{w}||} \tag{14.3}$$

where $p_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}/||\mathbf{w}||$ is the projection of any point $\mathbf{x} \in H_0$ on the decision

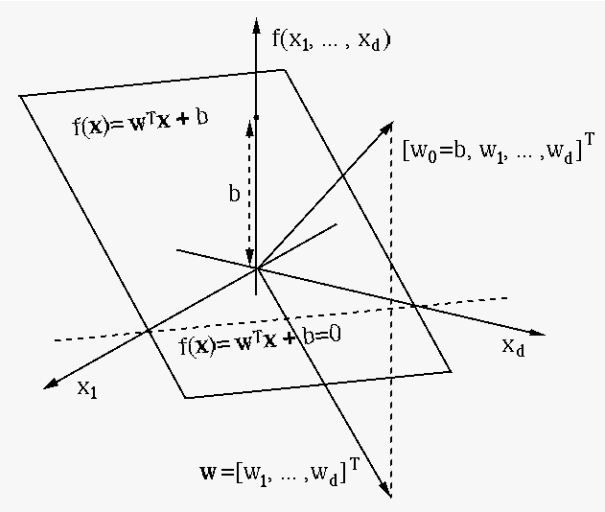


Figure 14.1 Binary Partition by Thresholding a Linear Function

plane H_0 onto its normal direction $\mathbf{w}$. The absolute value of this projection is the distance between H_0 and the origin:

$$d(H_0,\, \mathbf{0}) = |p_{\mathbf{w}}(\mathbf{x})| = \frac{|b|}{||\mathbf{w}||} \tag{14.4}$$

We further find the projection of any point $\mathbf{x}' \notin H_0$ off the decision plane H_0 onto $\mathbf{w}$ as $p_{\mathbf{w}}(\mathbf{x}') = \mathbf{w}^T\mathbf{x}'/||\mathbf{w}||$, and its distance to H_0 is:

$$d(H_0,\, \mathbf{x}') = \left| p_{\mathbf{w}}(\mathbf{x}') - p_{\mathbf{w}}(\mathbf{x}) \right| = \left| \frac{\mathbf{w}^T\mathbf{x}'}{||\mathbf{w}||} - \left(-\frac{b}{||\mathbf{w}||} \right) \right| = \frac{|\mathbf{w}^T\mathbf{x}' + b|}{||\mathbf{w}||} = \frac{|f(\mathbf{x}')|}{||\mathbf{w}||} \tag{14.5}$$

Fig. 14.2 shows the projections of 2-D points, $\mathbf{x} \in H_0$, together with $\mathbf{x}'$ and $\mathbf{x}''$ on opposite sides of H_0, onto its normal direction $\mathbf{w}$.

We desire to find the optimal decision plane H_0 that separates the training samples belonging to the two different classes, assumed to be linearly separable, in such a way that its distance to the support vectors, denoted by $\mathbf{x}_{sv}$, is maximized:

$$d(H_0,\, \mathbf{x}_{sv} \in C_+) = d(H_0,\, \mathbf{x}_{sv} \in C_-) = d(H_0,\, \mathbf{x}_{sv}) = \frac{|f(\mathbf{x}_{sv})|}{||\mathbf{w}||} \tag{14.6}$$

We further consider two additional planes H_+ and H_- that are in parallel with H_0 and pass through the support vectors $\mathbf{x}_{sv}$ on either side of H_0:

$$f_{\pm}(\mathbf{x}) = f(\mathbf{x}) \pm c = \mathbf{w}^T\mathbf{x} + b \pm c = \mathbf{w}^T\mathbf{x} + b \pm 1 = 0 \tag{14.7}$$

Here for convenience we have set $c = 1$ as these two equations can be arbitrarily scaled. Based on Eq. (14.4), the distances from H_+ or H_- to the origin can be

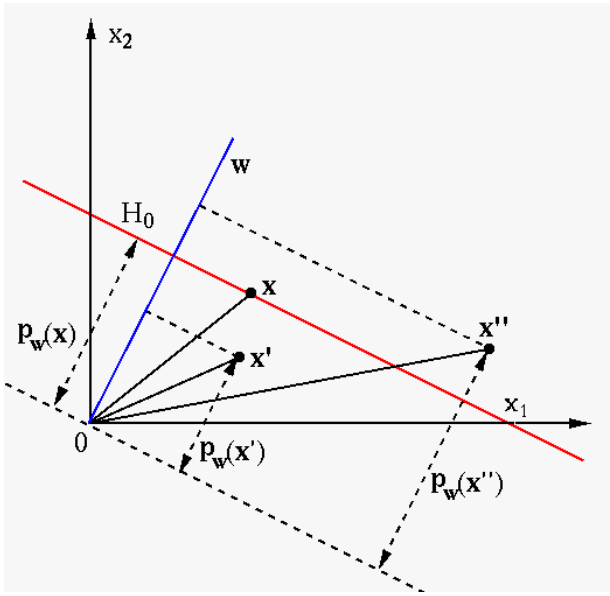


Figure 14.2 Decision Plane and Decision Margin

written as

$$d(H_{\pm}, \mathbf{0}) = \frac{|b \pm 1|}{||\mathbf{w}||} \tag{14.8}$$

and their distances to the decision plane H_0, called the decision *margin*, can be found as:

$$d(H_{\pm}, H_0) = \Big| d(H_{\pm}, \mathbf{0}) - d(H_0, \mathbf{0}) \Big| = \left| \frac{|b \pm 1|}{||\mathbf{w}||} - \frac{|b|}{||\mathbf{w}||} \right| = \frac{1}{||\mathbf{w}||} \tag{14.9}$$

To maximize this margin, we need to minimize $||\mathbf{w}||$.

For these planes to correctly separate all samples in the training set $\{(\mathbf{x}_n, y_n) \ (n = 1, \cdots, N)\}$, they need to satisfy the following two conditions:

$$\begin{cases} f_-(\mathbf{x}_n) = \mathbf{w}^T \mathbf{x}_n + b - 1 \geq 0 & \text{if } y_n = 1 \\ f_+(\mathbf{x}_n) = \mathbf{w}^T \mathbf{x}_n + b + 1 \leq 0 & \text{if } y_n = -1 \end{cases} \qquad (n = 1, \cdots, N) \tag{14.10}$$

where the equality is satisfied by the support vectors on H_+ or H_-, while the inequalities are satisfied by all other samples farther away from H_0 behind H_+ or H_-. The two conditions above can be combined to become:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \qquad (n = 1, \cdots, N) \tag{14.11}$$

Now the task of finding the optimal decision plane H_0 can be formulated as a constrained minimization problem:

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}||\mathbf{w}||^2 = \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ \text{subject to:} \quad & y_n(\mathbf{x}_n^T \mathbf{w} + b) \geq 1, \quad (n = 1, \cdots, N) \end{aligned} \tag{14.12}$$

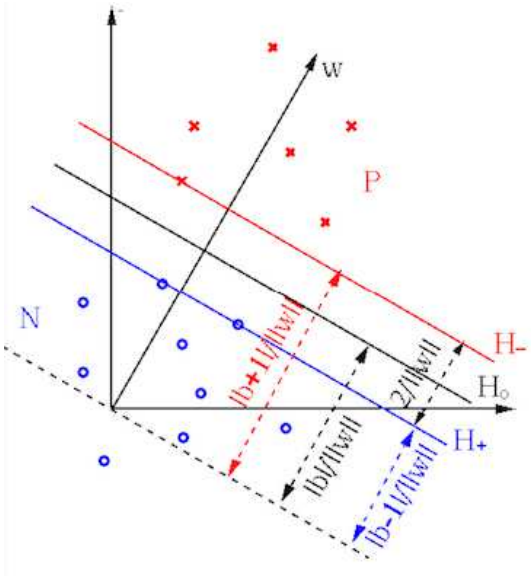


Figure 14.3 Decision Plane and Margin

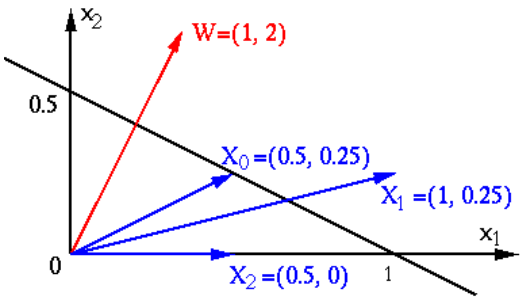


Figure 14.4 Decision Plane and Support Vectors

Fig. 14.3 illustrates the three planes considered above in 2-D case, including the decision plane H_0, together with H_+ and H_-, as well as the decision margin $1/||\mathbf{w}||$.

Example 14.1 In Fig. 14.4, the straight line in 2-D space, denoted by H_0, is described by the following linear equation

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = [w_1, w_2]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b = [1, 2]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - 1 = x_1 + 2x_2 - 1 = 0 \tag{14.13}$$

The distance from H_0 to the origin is:

$$d(H_0, \mathbf{0}) = \frac{|b|}{||\mathbf{w}||} = \frac{1}{\sqrt{w_1^2 + w_2^2}} = \frac{1}{\sqrt{1^2 + 2^2}} = \frac{1}{\sqrt{5}} = 0.447 \tag{14.14}$$

which is invariant with respect to any scaling of the equation. Consider three points in the space:

- $\mathbf{x}_0 = [0.5,\ 0.25]^T$, $f(\mathbf{x}_0) = 0.5 + 2 \times 0.25 - 1 = 0$, i.e., $\mathbf{x}_0$ is on the plane. Its distance to the plane is $d(\mathbf{x}_0, H) = f(\mathbf{x}_0)/||\mathbf{w}|| = 0$.
- $\mathbf{x}_1 = [1,\ 0.25]^T$, $f(\mathbf{x}_1) = 1 + 2 \times 0.25 - 1 = 0.5 > 0$, i.e., $\mathbf{x}_1$ is above the straight line, its distance to the plane is $d(\mathbf{x}_1, H) = f(\mathbf{x}_1)/||\mathbf{w}|| = |0.5|/\sqrt{5} = 0.2235$.
- $\mathbf{x}_2 = [0.5,\ 0]^T$, $f(\mathbf{x}_2) = 0.5 + 2 \times 0 - 1 = -0.5 < 0$, i.e., $\mathbf{x}_2$ is below the straight line, its distance to the plane is $d(\mathbf{x}_2, H) = f(\mathbf{x}_2)/||\mathbf{w}|| = |-0.5|/\sqrt{5} = 0.2235$.

As the two points $\mathbf{x}_1 = [1, 0.25]^T$ and $\mathbf{x}_2 = [0.5, 0]^T$ have equal distance to H_0, the straight line $f(\mathbf{x}) = x_1 + 2x_2 - 1 = 0$, we assume they are two support vectors $\mathbf{x}_{sv}$ on either side of H_0:

$$|f(\mathbf{x}_{sv})| = |\mathbf{w}^T\mathbf{x}_1 + b| = |\mathbf{w}^T\mathbf{x}_2 + b| = 0.5 \tag{14.15}$$

Based on the decision equation scaled by 2:

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 2x_1 + 4x_2 - 2 = 0 \tag{14.16}$$

we further get the equations for the two planes H_+ and H_- both parallel o H_0:

$$H_+ :\ 2x_1 + 4x_2 - 2 - 1 = 2x_1 + 4x_2 - 3 = 0$$
$$H_- :\ 2x_1 + 4x_2 - 2 + 1 = 2x_1 + 4x_2 - 1 = 0$$

Their distances to the origin are

$$d(H_+, \mathbf{0}) = \frac{|-3|}{||\mathbf{w}||} = 1.341, \quad d(H_-, \mathbf{0}) = \frac{|-1|}{||\mathbf{w}||} = 0.447 \tag{14.17}$$

and the distance between them is

$$d(H_-, H_+) = d(H_+, \mathbf{0}) - d(H_-, \mathbf{0}) 1.341 - 0.447 = 0.894 = \frac{2}{||\mathbf{w}||} \tag{14.18}$$

which is indeed twice the distance $d(H_\pm, H_0) = 1/||\mathbf{w}|| = 0.447$:

For reasons to be discussed later, instead of directly solving the constrained minimization problem in Eq. (14.12), now called the *primal problem*, we actually solve the *dual problem* (Section 3.3). We first construct the Lagrangian function of the primal problem, called the *primal function*:

$$L_p(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{n=1}^{N} \alpha_n (1 - y_n(\mathbf{w}^T\mathbf{x}_n + b)) \tag{14.19}$$

where $\alpha_1, \cdots, \alpha_N$ are the *Lagrange multipliers*. For this minimization problem

with non-positive constraints, the Lagrangian multipliers are required to be negative, α_n, according to Table 3.21 in Section 3.2, if a minus sign is used for the second term. However, to be consistent with all SVM literatures, we use the positive sign for the second term and require $\alpha_n \geq 0$. Note that if $\mathbf{x}_n$ is a support vector on either H_+ or H_-, i.e., the equality constraint $y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$ holds, then $\alpha_n > 0$; but if $\mathbf{x}_n$ is not a support vector, the equality constraint does not hold, and $\alpha_n = 0$.

As discussed in Subsection 3.3, we can solve this constrained optimization problem by solving an equivalent dual problem, i.e., we first find the lower bound of $L_p(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to $\mathbf{w}$ and b, and then find the tightest bound with respect to $\boldsymbol{\alpha}$). Such a solution has to satisfy the KKT conditions (Eqs. (3.45) through (3.48):

- Stationarity:

$$\begin{aligned}\frac{\partial}{\partial b}L_p(\mathbf{w}, b) &= \frac{\partial}{\partial b}\left[\frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{n=1}^{N}\alpha_n(1 - y_n(\mathbf{w}^T\mathbf{x}_n + b))\right] \\ &= \sum_{n=1}^{N}\alpha_n y_n = 0 \end{aligned} \tag{14.20}$$

and

$$\begin{aligned}\frac{\partial}{\partial \mathbf{w}}L_p(\mathbf{w}, b) &= \frac{\partial}{\partial \mathbf{w}}\left[\frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{n=1}^{N}\alpha_n(1 - y_n(\mathbf{w}^T\mathbf{x}_n + b))\right] \\ &= \mathbf{w} - \sum_{n=1}^{N}\alpha_n y_n \mathbf{x}_n = \mathbf{0}, \end{aligned} \tag{14.21}$$

i.e.,

$$\mathbf{w} = \sum_{n=1}^{N}\alpha_n y_n \mathbf{x}_n \tag{14.22}$$

- Primal feasibility:

$$y_n(\mathbf{w}^T\mathbf{x} + b) - 1 \geq 0, \qquad (n = 1, \cdots, N) \tag{14.23}$$

- Dual feasibility:

$$\alpha_n \geq 0, \qquad (n = 1, \cdots, N) \tag{14.24}$$

- Complementarity:

$$\alpha_n[y_n(\mathbf{w}^T\mathbf{x} + b) - 1] = 0 \tag{14.25}$$

To find the minimum as the lower bound of the primal function $L_p(\mathbf{w}, b)$ in Eq. (14.19), we substitute Eqs. (14.20) and (14.22) back into the primal function,

we get the *dual function*:

$$\begin{aligned} L_d(\boldsymbol{\alpha}) &= \min_{\mathbf{w},b} L_p(\mathbf{w}, b, \boldsymbol{\alpha}) = \min_{\mathbf{w},b} \left[\frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{n=1}^{N} \alpha_n (1 - y_n(\mathbf{w}^T\mathbf{x}_n + b)) \right] \\ &= \min_{\mathbf{w},b} \left[\frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{n=1}^{N} \alpha_n - \mathbf{w}^T \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n - b \sum_{n=1}^{N} \alpha_n y_n \right] \\ &= \frac{1}{2} \left(\sum_{m=1}^{N} \alpha_m y_m \mathbf{x}_m \right)^T \left(\sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n \right) \\ &\quad + \sum_{n=1}^{N} \alpha_n - \left(\sum_{m=1}^{N} \alpha_m y_m \mathbf{x}_m \right)^T \left(\sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n \right) \\ &= \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \left(\mathbf{x}_n^T \mathbf{x}_m \right) \end{aligned} \tag{14.26}$$

For this dual function $L_d(\boldsymbol{\alpha})$ to be the tightest lower bound of the primal function, it needs to be maximized with respect to $\boldsymbol{\alpha}$, subject to the constraint imposed by Eq. (14.20):

$$\begin{aligned} &\text{maximize:} \quad L_d(\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \left(\mathbf{x}_n^T \mathbf{x}_m \right) = \mathbf{1}^T \boldsymbol{\alpha} - \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{Q} \boldsymbol{\alpha} \\ &\text{subject to:} \quad \sum_{n=1}^{N} \alpha_n y_n = \mathbf{y}^T \boldsymbol{\alpha} = 0, \quad \alpha_n \geq 0 \quad (n = 1, \cdots, N) \end{aligned} \tag{14.27}$$

where $\mathbf{Q}$ is an N by N symmetric matrix of which the component in the mth row and nth column is $Q(m,n) = y_m y_n \mathbf{x}_m^T \mathbf{x}_n$ $(m, n = 1, \cdots, N)$. Now we have converted the primal problem of constrained minimization of the primal function $L_p(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to $\mathbf{w}$ and b to its dual problem of linearly constrained maximization of the dual function $L_d(\boldsymbol{\alpha})$ with respect to $\{\alpha_1, \cdots, \alpha_N\}$. Solving this quadratic programming (QP) problem (Section 3.6) we get all Lagrange multipliers $\alpha_n \geq 0$, $(n = 1, \cdots, N)$. All training samples $\mathbf{x}_n$ corresponding to positive Lagrange multipliers $\alpha_n > 0$ are support vectors, while others corresponding to $\alpha_n = 0$ are not support vectors.

We can now find the solution $\mathbf{w}^*$ based on Eq. (14.22):

$$\mathbf{w}^* = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n = \sum_{n \in sv} \alpha_n y_n \mathbf{x}_n = \sum_{x_n \in C_+,\, n \in sv} \alpha_n \mathbf{x}_n - \sum_{x_n \in C_-,\, n \in sv} \alpha_n \mathbf{x}_n \tag{14.28}$$

Note that the normal vector is the difference between the weighted means of the support vectors in class C_+ (first term) and class C_- (second term), and it is determined only by the support vectors.

We can further find b^* based on Eq. (14.25) for complementarity of the KKT conditions. As the Lagrangian multiplier α_n corresponding to any support vector

$\mathbf{x}_n$ is non-zero, we must have $y_n(\mathbf{w}^T\mathbf{x}_n+b)-1=0$, which is actually the equality of Eq. (14.11) satisfied by all support vectors:

$$y_n\left(\mathbf{w}^T\mathbf{x}_n+b\right)=1 \quad \text{or} \quad \mathbf{w}^T\mathbf{x}_n+b=y_n, \qquad (n\in sv) \tag{14.29}$$

(as $y_n^2=1$). Solving for b we get the optimal b^*:

$$b^*=y_n-\mathbf{w}^T\mathbf{x}_n=y_n-\sum_{m\in sv}\alpha_m y_m(\mathbf{x}_m^T\mathbf{x}_n), \qquad (n\in sv) \tag{14.30}$$

All support vectors should yield the same result. Computationally we simply get b^* as the average of the above for all support vectors.

We note that both $\mathbf{w}$ and b depend only on the support vectors on planes H_+ and H_-, corresponding to a positive $\alpha_n>0$, while all remaining samples corresponding to $\alpha_n=0$ are farther away from the decision plane H_0, behind either H_+ or H_-. We see that the SVM is solely based on those support vectors in the training set, once they are identified during the training process, all other non-support vectors are irrelevant.

Once $\mathbf{w}$ and b are obtained in the training process, any unlabeled sample $\mathbf{x}$ can be classified into either of the two classes depending on whether the following decision function is greater or smaller than zero:

$$\begin{aligned} f(\mathbf{x})=\mathbf{w}^T\mathbf{x}+b&=\left(\sum_{n\in sv}\alpha_n y_n\mathbf{x}_n\right)^T\mathbf{x}+b \\ &=\sum_{n\in sv}\alpha_n y_n\left(\mathbf{x}_n^T\mathbf{x}\right)+b \quad \begin{cases} >0 & \mathbf{x}\in C_+ \\ <0 & \mathbf{x}\in C_- \end{cases} \end{aligned} \tag{14.31}$$

We also get:

$$\begin{aligned} ||\mathbf{w}||^2=\mathbf{w}^T\mathbf{w}&=\left(\sum_{n\in sv}\alpha_n y_n\mathbf{x}_n^T\right)\left(\sum_{m\in sv}\alpha_m y_m\mathbf{x}_m\right) \\ &=\sum_{n\in sv}\alpha_n y_n\sum_{m\in sv}\alpha_m y_m(\mathbf{x}_n^T\mathbf{x}_m) \\ &=\sum_{n\in sv}\alpha_n y_n(y_n-b)=\sum_{n\in sv}\alpha_n(1-y_n b) \\ &=\sum_{n\in sv}\alpha_n-b\sum_{n\in sv}\alpha_n y_n=\sum_{n\in sv}\alpha_n \end{aligned} \tag{14.32}$$

The last equality is due to equality constraints of the dual problem $\sum_{n=1}^N\alpha_n y_n=0$. Now the decision margin can be written as:

$$\frac{1}{||\mathbf{w}||}=\left(\sum_{n\in sv}\alpha_n\right)^{-1/2} \tag{14.33}$$

The SVM algorithm for binary classification can now by summarized as the following steps:

- Find the Lagrange multipliers $\{\alpha_1, \cdots, \alpha_N\}$ by solving the QP problem in Eq. (14.27);
- Find the normal vector $\mathbf{w}$ by Eq. (14.28);
- Find the bias b by Eq. (14.30);
- Classify unlabeled $\mathbf{x}$ by Eq. (14.31).

As the last expression of the decision function in Eq. (14.31) depends only on α_n as well as $\mathbf{x}_n$ and y_n in the training set, the normal vector $\mathbf{w}$ is never needed, and the second step above for calculating $\mathbf{w}$ by Eq. (14.28) can be dropped.

We prefer to solve this dual problem not only because it is easier than the original primal problem, but also, more importantly, for the reason that all data points appear only in the form of an inner product $\mathbf{x}_n^T \mathbf{x}_m$ in the dual problem, allowing the *kernel method* to be used, as discussed later.

The Matlab code for the essential part of the algorithm is listed below, where `X` and `y` are respectively the data array composed of N training vectors $[\mathbf{x}_1, \cdots, \mathbf{x}_N]$ and their corresponding labelings $y_1, \cdots, y_N$, and `IPmethod` is a function that implements the *interior point method* (Section 3.7) for solving a general QP problem of minimizing $\mathbf{x}^T\mathbf{Q}\mathbf{x}/2 + \mathbf{c}^T\mathbf{x}$ subject to the linear equality constraints $\mathbf{A}\mathbf{x} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$.

```
[X y]=getData;       % get the training data
[m n]=size(X);
Q=(y'*y).*(X'*X);    % compute the quadratic matrix
c=-ones(n,1);        % coefficients of linear term
A=y;                 % coefficient matrix
b=0;                 % constants of linear equality constraints
alpha=0.1*ones(n,1); % initial guess of solution

[alpha mu lambda]=IPmethod(Q,A,c,b,alpha);
                     % solve QP for alpha (interior point method)
I=find(abs(alpha)>10^(-3));  % indices of non-zero alphas
asv=alpha(I);        % non-zero alphas
Xsv=X(:,I);          % support vectors
ysv=y(:,I);          % their labels
w=sum(repmat(ysv.*asv,m,1).*Xsv,2);  % normal vector (not needed)
bias=mean(ysv-w'*Xsv);               % bias
```

Example 14.2 The training set contains two classes of 2-D points with Gaussian distributions generated based on the following mean vectors and covariance matrices:

$$\mathbf{m}_- = \begin{bmatrix} 3.5 \\ 0 \end{bmatrix}, \quad \mathbf{S}_- = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}, \quad \mathbf{m}_+ = \begin{bmatrix} 0 \\ 3.5 \end{bmatrix}, \quad \mathbf{S}_+ = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The SVM algorithm finds the weight vector $\mathbf{w}$ and constant b of the optimal boundary between the two classes based on three support vectors, all listed

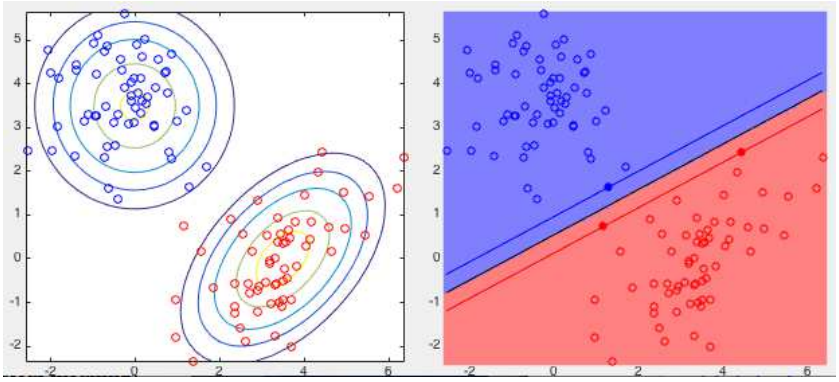


Figure 14.5 A Simple Example of Binary Classification by SVM

below. Note that decision boundary is completely dictated by the three support vectors and the margin distance between them is maximized.

n	α_n	$\mathbf{x}_n = [x_1, x_2]$	y_n
40	0.52	$x = [4.43, 2.44]$	-1
52	3.11	$x = [1.17, 0.74]$	-1
103	3.64	$x = [1.30, 1.64]$	1

, $\mathbf{w} = \begin{bmatrix} -1.25 \\ 2.39 \end{bmatrix}$, $b = -1.31$

The binary classification result is shown in Fig. 14.5.

14.2 Kernel SVM

The SVM algorithm above converges only if the data points of the two classes in the training set are linearly separable. If this is not the case, we can consider the *kernel method* first introduced in Section 11.1.

Kernel mapping can be applied to the SVM algorithm as all data points appear in the algorithm are in the form of an inner product. Specifically, during the training process, we replace the inner product $\mathbf{x}_m^T \mathbf{x}_n$ in both Eq. (14.27) and Eq. (14.30) by the kernel function $K(\mathbf{x}_m, \mathbf{x}_n)$ to get

$$\begin{aligned} \text{maximize:} \quad & L_d(\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m K(\mathbf{x}_n, \mathbf{x}_m) \\ \text{subject to:} \quad & \sum_{n=1}^{N} \alpha_n y_n = \mathbf{y}^T \boldsymbol{\alpha} = 0, \quad \alpha_n \geq 0 \quad (n = 1, \cdots, N) \end{aligned} \tag{14.34}$$

and

$$b = y_n - \sum_{m \in sv} \alpha_m y_m K(\mathbf{x}_m, \mathbf{x}_n), \qquad (n \in sv) \tag{14.35}$$

We also replace the inner product $\mathbf{x}_n^T\mathbf{x}$ in Eq. (14.31) by $K(\mathbf{x}_n, \mathbf{x})$ for the classification of any unlabeled point $\mathbf{z} = \phi(\mathbf{x})$:

$$f(\mathbf{z}) = \mathbf{w}^T\mathbf{z}+b = \sum_{n\in sv} \alpha_n y_n (\mathbf{z}_n^T\mathbf{z})+b = \sum_{n\in sv} \alpha_n y_n K(\mathbf{x}_n, \mathbf{x})+b \quad \begin{cases} >0 & \mathbf{x}\in C_+ \\ <0 & \mathbf{x}\in C_- \end{cases} \tag{14.36}$$

Again, we note that the normal vector $\mathbf{w}$ in Eq. (14.28) never needs to be explicitly calculated. As now both the training and classification are carried out in some higher dimensional space $\mathbf{z} = \phi(\mathbf{x})$, in which the classes are more likely linearly separable, the classification can be more effectively. More generally, the kernel method can be applied to any algorithm, so long as the data always appear in the form of an inner product.

14.3 Soft Margin SVM

When the two classes C_- and C_+ are not linearly separable, the condition for the optimal hyperplane in Eq. (14.11) can be relaxed by including a slack variable $\xi_n \geq 0$ representing an extra error term:

$$y_n(\mathbf{x}_n^T\mathbf{w} + b) \geq 1 - \xi_n, \quad (n = 1, \cdots, N) \tag{14.37}$$

For better separate data points belonging to different classes, this error ξ_n needs to be minimized as well as $||\mathbf{w}||$. Now the primal problem in Eq. (14.12) can be modified to the following minimization problem with an objective function and two sets of inequality (non-negative) constraints:

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{n=1}^{N}\xi_n \\ \text{subject to:} \quad & y_n(\mathbf{x}_n^T\mathbf{w} + b) + \xi_n - 1 \geq 0, \quad \xi_n \geq 0; \quad (n = 1, \cdots, N) \end{aligned} \tag{14.38}$$

The value of the parameter C can be adjusted to make a proper tradeoff between under fitting emphasizing the global distribution of the training samples, and over fitting emphasizing the local samples close to the decision boundary H_0. A large C value emphasizes the minimization of the error term in the objective function, and the resulting H_0 is mostly dictated by a small number of local support vectors in the region between the class, including possibly some noisy outliers, thereby risking the problem of overfitting. On the other hand, a small C value de-emphasizes the error term and the classifier can tolerate greater errors, thereby allowing more data samples farther away from H_0 to become support vectors and play a role in determining H_0. Consequently the classification trends to better reflect the overall distribution of the data samples, but with the risk of overfitting.

This constrained minimization problem can be converted to the minimization

of the corresponding Lagrangian:

$$L_p(\mathbf{w}, b, \xi, \boldsymbol{\alpha}, \mu) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{n=1}^{N}\xi_n - \sum_{n=1}^{N}\alpha_n[y_n(\mathbf{w}^T\mathbf{x}+b)+\xi_n-1] - \sum_{n=1}^{N}\beta_n\xi_n \tag{14.39}$$

where β_n as well as α_n is the Lagrange multipliers for the two sets of constraints. As they are non-negative (instead of non-positive as in previous section), minus sign is used in front of the last two terms for the constraints, so that the Lagrange multipliers α_n and β_n are still required to be positive (in consistent with Table 3.21). The KKT conditions (Eqs. (3.45) through (3.48)) of this minimization problem are listed below (for all $n = 1, \cdots, N$):

- Stationarity:

$$\frac{\partial L_p}{\partial \mathbf{w}} = \mathbf{w} - \sum_{n=1}^{N} y_n\alpha_n\mathbf{x}_n = 0, \quad \text{i.e.} \quad \mathbf{w} = \sum_{n=1}^{N} y_n\alpha_n\mathbf{x}_n; \tag{14.40}$$

$$\frac{\partial L_p}{\partial b} = \sum_{n=1}^{N} y_n\alpha_n = 0; \tag{14.41}$$

$$\frac{\partial L_p}{\partial \xi_n} = C - \alpha_n - \beta_n = 0; \tag{14.42}$$

- Primal feasibility:

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) + \xi_n - 1 \geq 0, \quad \xi_n \geq 0, \tag{14.43}$$

- Complementarity:

$$\alpha_n[y_n(\mathbf{w}^T\mathbf{x}_n + b) + \xi_n - 1] = 0, \quad \beta_n\xi_n = 0, \tag{14.44}$$

- Dual feasibility:

$$\alpha_n \geq 0, \quad \beta_n \geq 0 \tag{14.45}$$

Depending on the value of α_n, there exist three possible cases, due to the complementarity (Eq. (14.44)):

- If $\alpha_n = 0$, then $\beta_n = C - \alpha_n = C > 0$, $\xi_n = 0$, and we have

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1 \geq 0 \tag{14.46}$$

- If $0 < \alpha_n < C$, i.e., $\alpha_n \neq 0$, then we have

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) + \xi_n - 1 = 0 \tag{14.47}$$

  But as $\beta_n = C - \alpha_n > 0$, $\xi_n = 0$ and the equation above becomes

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1 = 0 \tag{14.48}$$

- If $\alpha_n = C \neq 0$, then we have

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) + \xi_n - 1 = 0 \tag{14.49}$$

But as $\beta_n = C - \alpha_n = 0$, $\xi_n \geq 0$ and the equation above becomes

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1 \leq 0 \tag{14.50}$$

In all three cases the expression $y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1$ can be rewritten as

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1 = y_n(\mathbf{w}^T\mathbf{x}_n + b) - y_n^2 = y_n[(\mathbf{w}^T\mathbf{x}_n + b) - y_n] = y_n\, E_n \tag{14.51}$$

where $E_n = (\mathbf{w}^T\mathbf{x}_n + b) - y_n$ is the error, or the difference between the actual and desired output.

We summarize all three cases above to get:

$$y_n\, E_n \begin{cases} \geq 0 & \text{if } \alpha_n = 0 \\ = 0 & \text{if } 0 < \alpha_n < C \\ \leq 0 & \text{if } \alpha_n = C \end{cases} \tag{14.52}$$

These results derived from the KKT conditions in Eqs. (14.40) through (14.45) are an alternative form of the KKT conditions, which will be used in SMO algorithm to check if the KKT conditions are violated by any α_n.

The dual of the primal problem in Eq. (14.39) can be obtained by substituting the first set of equations (stationarity) into the primal Lagrangian:

$$\begin{aligned} \text{maximize:} \quad & L_d(\boldsymbol{\alpha}) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m y_n y_m \mathbf{x}_m^T\mathbf{x}_n \\ & + C\sum_{n=1}^{N}\xi_n - \sum_{n=1}^{N}\alpha_n\xi_n - \sum_{n=1}^{N}\beta_n\xi_n \\ & = \mathbf{1}^T\boldsymbol{\alpha} - \frac{1}{2}\boldsymbol{\alpha}^T\mathbf{Q}\boldsymbol{\alpha} + C\sum_{n=1}^{N}\xi_n - \sum_{n=1}^{N}\alpha_n\xi_n - \sum_{n=1}^{N}\beta_n\xi_n \\ \text{subject to:} \quad & 0 \leq \alpha_n \leq C, \quad \sum_{n=1}^{N}\alpha_n y_n = 0 \end{aligned} \tag{14.53}$$

Note that due to the condition $C = \alpha_n + \beta_n$, the Lagrangian multipliers ξ_n and β_n no longer appear in the Lagrangian L_d of the dual problem, which happens to be identical to that of the linearly separable problem discussed in previous section. Also, since $\alpha_n \geq 0$, $\beta_n \geq 0$ and $C = \alpha_n + \beta_n$, we have $0 \leq \alpha_n = C - \beta_n \leq C$. Having solved this QP problem for $\alpha_1, \cdots, \alpha_N$, we get the optimal decision plane in terms of the normal vector of the decision plane based on Eq. (14.28):

$$\mathbf{w} = \sum_{n=1}^{N} y_n\alpha_n\mathbf{x}_n = \sum_{n\in sv} y_n\alpha_n\mathbf{x}_n \tag{14.54}$$

and the bias term b based on Eq. (14.30):

$$b = y_n - \sum_{i\in sv} y_i\alpha_i\mathbf{x}_i^T\mathbf{x}_n = y_n - \sum_{i\in sv} y_i\alpha_i K(\mathbf{x}_i, \mathbf{x}_n), \qquad (n \in sv) \tag{14.55}$$

where the inner product $\mathbf{x}_k^T\mathbf{x}$ is replaced by the kernel function $K(\mathbf{x}_k, \mathbf{x})$ if it is nonlinear. Computationally, we take the average of such b values of all support vectors as the offset term.

Once $\mathbf{w}$ and b are obtained, any unlabeled point $\mathbf{x}$ can be classified into either of the two classes depending on whether it is on the positive or negative side of the decision plane:

$$\mathbf{w}^T\mathbf{x} + b = \sum_{i \in sv} y_i \alpha_i \mathbf{x}_i^T \mathbf{x} + b = \sum_{i \in sv} y_i \alpha_i K(\mathbf{x}_i, \mathbf{x}) + b \begin{cases} > 0, & \mathbf{x} \in C_+ \\ < 0, & \mathbf{x} \in C_- \end{cases} \quad (14.56)$$

where the inner product is replaced by a kernel function if it is nonlinear.

14.4 Sequential Minimal Optimization Algorithm

The *sequential minimal optimization (SMO)* is a more efficient algorithm for solving the SVM problem, compared with the generic QP algorithms such as the interior-point method. The SMO algorithm can be considered as a method of decomposition, by which an optimization problem of multiple variables is decomposed into a series of sub-problems each optimizing an objective function of a small number of variables, typically only one, while all other variables are treated as constants that remain unchanged in the sub-problem. For example, the coordinate descent algorithm is just such a decomposition method, which solves a problem in a multidimensional space by converting it into a sequence of sub-problems each in a one-dimensional space.

When applying the decomposition method to the soft margin SVM problem, we could consider optimizing only one variable α_i at a time, while treating all remaining variables $\alpha_1, \cdots, \alpha_{i-1}, \alpha_{i+1}, \cdots, \alpha_N$ as constants. However, due to the constraint $\sum_{n=1}^N \alpha_n y_n = 0$, α_i is a linear combination of $N-1$ constants and is therefore also a constant. We therefore need to consider two variables at a time, denoted by α_i and α_j, while treating the remaining $N-2$ variables as constants. The objective function of such a sub-problem can be obtained by dropping all constant terms independent of the two selected variables α_i and α_j

in the original objective function in Eq. (14.53):

$$\begin{aligned}
\text{maximize:} \quad & L(\alpha_i, \alpha_j) = \alpha_i + \alpha_j - \frac{1}{2}\left(\alpha_i^2 \mathbf{x}_i^T \mathbf{x}_i + \alpha_j^2 \mathbf{x}_j^T \mathbf{x}_j + 2\alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j\right) \\
& -\alpha_i y_i \left(\sum_{n \neq i} \alpha_n y_n \mathbf{x}_n^T\right) \mathbf{x}_i - \alpha_j y_j \left(\sum_{n \neq j} \alpha_n y_n \mathbf{x}_n^T\right) \mathbf{x}_j \\
= & \alpha_i + \alpha_j - \frac{1}{2}\left(\alpha_1^2 K_{ii} + \alpha_2^2 K_{jj} + 2\alpha_i \alpha_j y_i y_j K_{ij}\right) \\
& -\alpha_i y_i \sum_{n \neq i,j} \alpha_n y_n K_{ni} - \alpha_j y_j \sum_{n \neq i,,j} \alpha_n y_n K_{nj} \\
\text{subject to:} \quad & 0 \leq \alpha_i, \alpha_j \leq C, \qquad \sum_{n=1}^{N} \alpha_n y_n = 0
\end{aligned} \tag{14.57}$$

Here all inner products $\mathbf{x}_m^T \mathbf{x}_n$ are replaced by a kernel the corresponding kernel function $K_{mn} = K(\mathbf{x}_m, \mathbf{x}_n)$.

This maximization problem can be solved iteratively. Out of all previous values α_n^{old} $(n = 1, \cdots, N)$, the two selected variables are updated to become α_i^{new} and α_j^{new} by which $L(\alpha_i, \alpha_j)$ is maximized subject to the two constraints.

We rewrite the second constraint as

$$\alpha_i y_i + \alpha_j y_j = -\sum_{n \neq i,j} \alpha_n y_n \tag{14.58}$$

and multiply both sides by y_i to get

$$y_i^2 \alpha_i + y_i y_j \alpha_j = \alpha_i + s\alpha_j = \left(-\sum_{n \neq i,j} \alpha_n y_n\right) y_i = \delta, \quad \text{i.e.} \quad \alpha_i = \delta - s\alpha_j \tag{14.59}$$

where $y_i^2 = 1$, and we have defined

$$s = y_i y_j, \qquad \delta = -\sum_{n \neq i,j} \alpha_n y_n y_i \tag{14.60}$$

As δ is independent of α_i and α_j, it remains the same before and after they are updated, and we have

$$\alpha_i^{new} + s\alpha_j^{new} = \alpha_i^{old} + s\alpha_j^{old} = \delta \tag{14.61}$$

i.e.,

$$\Delta\alpha_i = \alpha_i^{new} - \alpha_i^{old} = -s(\alpha_j^{new} - \alpha_j^{old}) = -s\Delta\alpha_j \tag{14.62}$$

We now consider a closed-form solution for updating α_i and α_j in each iteration of this two-variable optimization problem. We first rewrite Eq. (14.28) as

$$\sum_{n \neq i,j} \alpha_n y_n \mathbf{x}_n = \mathbf{w} - \alpha_i y_i \mathbf{x}_i - \alpha_j y_j \mathbf{x}_j \tag{14.63}$$

so that the two summations in $L(\alpha_i, \alpha_j)$, now denoted by v_k $(k = i, j)$, can be written as

$$\begin{aligned} v_k &= \sum_{n \neq i,j} \alpha_n y_n K_{nk} = \sum_{n=1}^{N} \alpha_n y_n K_{nk} - \alpha_i y_i K_{ik} - \alpha_j y_j K_{jk} \\ &= \left(\sum_{n=1}^{N} \alpha_n y_n K_{nk} + b \right) - b - \alpha_i y_i K_{ik} - \alpha_j y_j K_{jk} \\ &= u_k - b - \alpha_i y_i K_{ik} - \alpha_j y_j K_{jk} \end{aligned} \tag{14.64}$$

where

$$u_k = f(\mathbf{z}_k) = \sum_{n=1}^{N} \alpha_n y_n K_{nk} + b \tag{14.65}$$

is the output in Eq. (14.36) corresponding to input $\mathbf{x}_k$. Now the objective function above can be written as:

$$L(\alpha_i, \alpha_j) = \alpha_i + \alpha_j - \frac{1}{2}(\alpha_i^2 K_{ii} + \alpha_j^2 K_{jj} + 2s\alpha_i \alpha_j K_{ij}) - \alpha_i y_i v_i - \alpha_j y_j v_j \tag{14.66}$$

Substituting $\alpha_i = \delta - s\alpha_j$ (Eq. (14.62)) into $L(\alpha_i, \alpha_j)$, we can rewrite it as a function of a single variable α_j alone:

$$\begin{aligned} L(\alpha_j) &= \delta + (1-s)\alpha_j - \frac{1}{2}\left[(\delta - s\alpha_j)^2 K_{ii} + \alpha_j^2 K_{jj} + 2s(\delta - s\alpha_j)\alpha_j K_{ij}\right] \\ &\quad -(\delta - s\alpha_j) y_i v_i - \alpha_j y_j v_j \\ &= \frac{1}{2}(2K_{ij} - K_{ii} - K_{jj})\alpha_j^2 \\ &\quad +[1 - s + s\delta(K_{ii} - K_{ij}) + y_j(v_i - v_j)]\alpha_j + \text{Const} \end{aligned} \tag{14.67}$$

where $s^2 = (y_i y_j)^2 = 1$, and scalar Const represents all constant terms independent of α_i and α_j and can therefore be dropped. Now the objective function becomes a quadratic function of the single variable α_j. Consider the coefficients of this quadratic function:

- The coefficient of α_j^2:

$$\frac{1}{2}(2K_{ij} - K_{ii} - K_{jj}) = \frac{1}{2}\eta \tag{14.68}$$

  where η is defined as

$$\begin{aligned} \eta &= 2K_{ij} - K_{ii} - K_{jj} = 2\mathbf{z}_i^T \mathbf{z}_j - \mathbf{z}_i^T \mathbf{z}_i - \mathbf{z}_j^T \mathbf{z}_j = -(\mathbf{z}_i - \mathbf{z}_j)^T (\mathbf{z}_i - \mathbf{z}_j) \\ &= -||\mathbf{z}_i - \mathbf{z}_j||^2 \leq 0 \end{aligned} \tag{14.69}$$

- The coefficient of α_j:

$$\begin{aligned}
&1 - s + s\delta(K_{ii} - K_{ij}) + y_j(v_i - v_j) \\
&= 1 - s + (s\alpha_i + \alpha_j)(K_{ii} - K_{ij}) \\
&\quad + y_j[(u_i - b - \alpha_i y_i K_{ii} - \alpha_j y_j K_{ij}) - (u_j - b - \alpha_i y_i K_{ij} - \alpha_j y_j K_{jj})] \\
&= 1 - s + s\alpha_1(K_{ii} - K_{ij}) + \alpha_2(K_{ii} - K_{ij}) \\
&\quad + y_j(u_i - u_j) - \alpha_i s K_{ii} - \alpha_j K_{ij} + \alpha_i s K_{ij} + \alpha_j K_{jj} \\
&= y_j^2 - s - \alpha_j(2K_{ij} - K_{ii} - 2K_{jj}) + y_j(u_i - u_j) \\
&= y_j(u_i - y_j - u_j + y_j) - \alpha_j(2K_{ij} - K_{ii} - K_{jj}) \\
&= y_j(E_i - E_j) - \alpha_j \eta
\end{aligned} \tag{14.70}$$

  where we have defined

$$E_k = u_k - y_k = \sum_{n=1}^{N} \alpha_n^{old} y_n K_{nk} + b - y_k \qquad (k = 1, 2) \tag{14.71}$$

  as the difference between the desired output y_k and the actual output based on the previous values of the variables α_n^{old} $(n = 1, \cdots, N)$.

Now the quadratic objective function can be rewritten as:

$$L(\alpha_j) = \frac{1}{2}\eta\alpha_j^2 + \left[y_j(E_i - E_j) - \alpha_j^{old}\eta\right]\alpha_j \tag{14.72}$$

Given the previous values α_n^{old} $(n = 1, \cdots, N)$ in the coefficients, we will find α_j^{new} that maximizes $L(\alpha_j)$. Consider its first and second order derivatives:

$$\begin{aligned}
&\frac{d}{d\alpha_j} L(\alpha_j) = \eta\alpha_j + y_j(E_i - E_j) - \alpha_j^{old}\eta \\
&\frac{d^2}{d\alpha_j^2} L(\alpha_j) = \eta \leq 0
\end{aligned} \tag{14.73}$$

As the second order derivative of $L(\alpha_j)$ is $\eta \leq 0$, it has a maximum. Solving the equation $d\,L(\alpha_j)/d\alpha_j = 0$, we get α_j^{new} that maximizes $L(\alpha_j)$ based on the old value α_j^{old}:

$$\alpha_j^{new} = \alpha_j^{old} + \frac{y_j(E_j - E_i)}{\eta} = \alpha_j^{old} + \Delta\alpha_j, \qquad \Delta\alpha_j = \alpha_j^{new} - \alpha_j^{old} = \frac{y_j(E_j - E_i)}{\eta}, \tag{14.74}$$

Note that this is an unconstrained solution, which needs to be modified if the constraints in Eq. (14.57) are violated. As shown in Fig. 14.6, α_i and α_j are inside the square of size C, due to the first constraint $0 \leq \alpha_i,\ \alpha_j \leq C$; also, due to the second constraint $\alpha_i + s\alpha_j = \delta$ (Eq. (14.59)), they have to be on the diagonal line segment inside the square.

The panels in Fig. 14.6 illustrate the four possible cases depending on whether $s = 1$ ($y_i = y_j = \pm 1$, panels 1 and 2), or $s = -1$ ($y_i = -y_j = \pm 1$, panels 3 and 4), and whether $\delta < C$ (panels 1 and 3) or $\delta > C$ (panels 2 and 4), which can

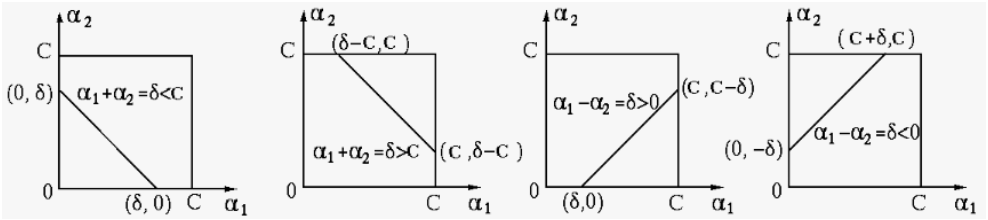


Figure 14.6 Four Cases while Updating α_i and α_i

be further summarized below in terms of the lower bound L and upper bound H of α_2:

- If $s = 1$, then $\alpha_i + s\alpha_j = \alpha_i + \alpha_j = \delta$,
  - if $\delta < C$, then $\min(\alpha_j) = 0,\ \max(\alpha_j) = \delta$ (first panel)
  - if $\delta > C$, then $\min(\alpha_j) = \delta - C,\ \max(\alpha_j) = C$ (second panel)

  Combining these two cases, we have

$$\begin{cases} L = \max(0,\ \delta - C) = \max(0,\ \alpha_i + \alpha_j - C) \\ H = \min(\delta,\ C) = \min(\alpha_i + \alpha_j,\ C) \end{cases} \tag{14.75}$$

- If $s = -1$, then $\alpha_i + s\alpha_j = \alpha_i - \alpha_j = \delta$,
  - if $\delta > 0$, then $\min(\alpha_j) = 0,\ \max(\alpha_j) = C - \delta$ (third panel)
  - if $\delta < 0$, then $\min(\alpha_j) = -\delta,\ \max(\alpha_j) = C$ (fourth panel)

  Combining these two cases, we have

$$\begin{cases} L = \max(0,\ -\delta) = \max(0,\ \alpha_j - \alpha_i) \\ H = \min(C - \delta,\ C) = \min(C + \alpha_j - \alpha_i,\ C) \end{cases} \tag{14.76}$$

Now we apply the constraints in terms of L and H to t optimal solution α_j^{new} obtained previously to get α_j that is feasible as well as optimal:

$$\alpha_j^{new} \Longleftarrow \begin{cases} H & \text{if } \alpha_j^{new} \geq H \\ \alpha_j^{new} & \text{if } L < \alpha_2^{new} < H \\ L & \text{if } \alpha_j^{new} \leq L \end{cases} \tag{14.77}$$

We can further find α_i based on Eq. (14.62):

$$\alpha_i^{new} = \alpha_i^{old} - s\Delta\alpha_j = \alpha_i^{old} - s(\alpha_j^{new} - \alpha_j^{old}) \tag{14.78}$$

and update the weight vector based on Eq. (14.40):

$$\begin{aligned} \mathbf{w}^{new} &= \sum_{n \neq i,j} y_n \alpha_n^{old} \mathbf{x}_n + y_i \alpha_i^{new} \mathbf{x}_i + y_j \alpha_j^{new} \mathbf{x}_j \\ &= \mathbf{w}^{old} - y_i \alpha_i^{old} \mathbf{x}_i - y_j \alpha_j^{old} \mathbf{x}_j + y_i \alpha_i^{new} \mathbf{x}_i + y_j \alpha_j^{new} \mathbf{x}_j \\ &= \mathbf{w}^{old} + y_i \Delta\alpha_i \mathbf{x}_i + y_j \Delta\alpha_j \mathbf{x}_j = \mathbf{w}^{old} + \Delta\mathbf{w} \end{aligned} \tag{14.79}$$

where $\Delta \mathbf{w} = \mathbf{w}^{new} - \mathbf{w}^{old} = y_i \Delta\alpha_i \mathbf{x}_i + y_j \Delta\alpha_j \mathbf{x}_j$. To update b, consider:

$$\begin{aligned}\Delta E_k &= E_k^{new} - E_k^{old} = u_k^{new} - u_k^{old} \\ &= \sum_{n=1}^{N} \alpha_n^{new} y_n K_{nk} + b^{new} - \sum_{n=1}^{N} \alpha_n^{old} y_n K_{nk} - b^{old} \\ &= y_i \Delta\alpha_i K_{ik} + y_j \Delta\alpha_j K_{jk} + b^{new} - b^{old} \quad (k = 1, 2) \end{aligned} \tag{14.80}$$

Consider the following cases:

- If $0 < \alpha_k < C$ for either $k = i$ or $k = j$, then $y_k E_k = 0$ according to Eq. (14.52). We let $E_k^{new} = 0$ and solve the equation above to get:

$$b_k^{new} = b_k^{old} - (E_k^{old} + y_i \Delta\alpha_i K_{ik} + y_j \Delta\alpha_j K_{jk}) \quad (k = 1 \text{ or } 2) \tag{14.81}$$

- If $0 < \alpha_k < C$ for both $k = i$ and $k = j$, then $b_i^{new} = b_j^{new}$.
- If $\alpha_k = 0$ or $\alpha_k = C$ for both $k = i$ and $k = 2$, then $E_k \neq 0$, and $b_i^{new} \neq b_j^{new}$, their average $b^{new} = (b_i^{new} + b_j^{new})/2$ can be used.

The computation above for maximizing $L(\alpha_i, \alpha_j)$ is carried out iteratively each time when a pair of variables α_i and α_j is selected to be updated. Specifically, we select a variable α_i that violates any of the KKT conditions given in Eq. (14.52) in the outer loop of the SMO algorithm, and then pick a second variable α_j in the inner loop of the algorithm, both to be optimized. The selection of these variables can be either random (e.g., by following their order), or guided by some heuristics, such as always choosing the variable with maximum step size $\Delta\alpha_n = \alpha_n^{new} - \alpha_n^{old}$. This iterative process is repeated until convergence when the KKT conditions are satisfied by all $\alpha_1, \cdots, \alpha_N$ variables, i.e., no more variables need to be updated. All data points corresponding to $\alpha_n \neq 0$ are the support vectors, based on which we can find the offset b by Eq. (14.55) and classify any unlabeled point $\mathbf{x}$ by Eq. (14.56).

The Matlab code for the essential part of the SMO algorithm is listed below, based mostly on a simplified SMO algorithm with some modifications.

In particular, the if-statement

```
if (alpha(i)<C & yE<-tol) | (alpha(i)>0 & yE>tol)
```

checks the three KKT conditions in Eq. (14.52), where $yE = y_n E_n$ and *tol* is a small constant. The first half checks if the first two of the three conditions are violated, while the second half checks if the second two of the three conditions are violated.

```
[X y]=getData;          % get training data
[m n]=size(X);          % size of data
alpha=zeros(1,n);       % alpha variables
bias=0;                 % initial bias
it=0;                   % iteration index
```

```
while (it<maxit)        % number of iterations less than maximum
    it=it+1;
    changed_alphas=0;   % number of changed alphas
    N=length(y);        % number of support vectors
    for i=1:N           % for each alpha_i
        Ei=sum(alpha.*y.*K(X,X(:,i)))+bias-y(i);
        yE=Ei*y(i);
        if (alpha(i)<C & yE<-tol) | (alpha(i)>0 & yE>tol)
                        % KKT violation
            for j=[1:i-1,i+1:N]   % for each alpha_j unequal alpha_i
                Ej=sum(alpha.*y.*K(X,X(:,j)))+bias-y(j);
                ai=alpha(i);      % alpha_i old
                aj=alpha(j);      % alpha_j old
                if y(i)==y(j)     % s=y_i y_j=1
                    L=max(0,alpha(i)+alpha(j)-C);
                    H=min(C,alpha(i)+alpha(j));
                else              % s=y_i y_j=-1
                    L=max(0,alpha(j)-alpha(i));
                    H=min(C,C+alpha(j)-alpha(i));
                end
                if L==H           % skip when L==H
                    continue
                end
                eta=2*K(X(:,j),X(:,i))-K(X(:,i),X(:,i))-K(X(:,j),X(:,j));
                alpha(j)=alpha(j)+y(j)*(Ej-Ei)/eta;   % update alpha_j
                if alpha(j) > H
                    alpha(j) = H;
                elseif alpha(j) < L
                    alpha(j) = L;
                end
                if norm(alpha(j)-aj) < tol   % skip if no change
                    continue
                end
                alpha(i)=alpha(i)-y(i)*y(j)*(alpha(j)-aj);
                                                    % find alpha_i
                bi = bias - Ei - y(i)*(alpha(i)-ai)*K(X(:,i),X(:,i))...
                    -y(j)*(alpha(j)-aj)*K(X(:,j),X(:,i));
                bj = bias - Ej - y(i)*(alpha(i)-ai)*K(X(:,i),X(:,j))...
                    -y(j)*(alpha(j)-aj)*K(X(:,j),X(:,j));
                if 0<alpha(i) & alpha(i)<C
                    bias=bi;
                 elseif 0<alpha(j) & alpha(j)<C
                    bias=bj;
                else
```

```
                    bias=(bi+bj)/2;
                end
                changed_alphas=changed_alphas+1;  % one more alpha changed
            end
        end
    end
    if changed_alphas==0      % no more changed alpha, quit
        break
    end
    I=find(alpha~=0);         % indecies of non-zero alphas
    alpha=alpha(I);           % find non-zero alphas
    Xsv=X(:,I);               % find support vectors
    ysv=y(I);                 % their corresponding indecies
end                           % end of iteration
nsv=length(ysv);              % number of support vectors
```

Here `K(X,x)` is a function that takes an $m \times n$ matrix $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_n]$ and an n-D vector $\mathbf{x}$ and produces the kernel functions $K(\mathbf{x}_i, \mathbf{x}),\ (i = 1, \cdots, n)$ as output.

```
function ker=K(X,x)
    global kfunction
    global gm                  % parameter for Gaussian kernel
    [m n]=size(X);
    ker=zeros(1,n);
    if kfunction=='l'
        for i=1:n
            ker(i)=X(:,i).'*x; % linear kernel
        end
    elseif kfunction=='g'
        for i=1:n
            ker(i)=exp(-gm*norm(X(:,i)-x)); % Gaussian kernel
        end
    elseif kfunction=='p'
        for i=1:n
            ker(i)=(X(:,i).'*x).^3; % polynomial kernel
        end
    end
end
```

Having found all non-zero $\alpha_n \neq 0$ and the corresponding support vectors, we further find the offset or bias term:

```
 bias=0;
 for i=1:nsv
    bias=bias+(ysv(i)-sum(ysv.*alpha.*K(Xsv,Xsv(:,i))));
```

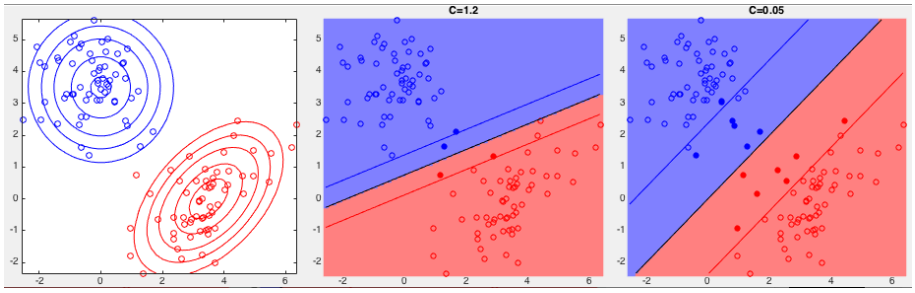


Figure 14.7 Classification by SVM Based on SMO

```
end
bias=bias/nsv;
```

Any unlabeled data point $\mathbf{x}$ is classified into class C_+ or C_-, depending on whether the following expression is greater or smaller than zero:

```
y=sum(alpha.*ysv.*K(Xsv,x))+bias;
```

Example 14.3 The same training set used previously to test the SVM with a hard margin is used again to test the SVM with a soft margin. Two results are shown in Fig. 14.7, corresponding to two different values $C = 1.2$ and $C = 0.05$ for the weight of the error term $\sum_{n=1}^{N} \xi_n$ in the objective function. We see that when $C = 1.2$ is large, the number of support vectors (the four solid dots) is small due to the small errors ξ_n allowed, and the decision boundary determined by the these support vectors independent of the rest of the data set (circles) is mostly dictated by the local points close to the boundary, not necessarily a good reflection of the global distribution of the data points. This result is similar to the previous one generated by the hard-margin SVM.

On the other hand, when $C = 0.1$ is small, a greater number of data points become support vectors due to the larger (the 13 solid dots) error ξ_n allowed, and the resulting linear decision line determined by these support vectors reflects global distribution of the data points, and it better separates the two classes in terms of their mean positions.

Example 14.4 The classification results for the XOR data set are shown in Fig. 14.8 with $C = 1.2$ and $C = 0.05$. As the two classes in the XOR data set are not linearly separable, a Gaussian or radial basis function (RBF) kernel is used to map the data from 2-D space to an infinite dimensional space, which is partitioned into two regions for the two classes. When $C = 1.2$ is large, the decision boundary is dictated by a relatively small number of support vectors (solid dots, with small error ξ_n) and all data points are classified correctly, but it is more prone to the overfitting problem, if some of the small number of support vectors are outliers due to noise.

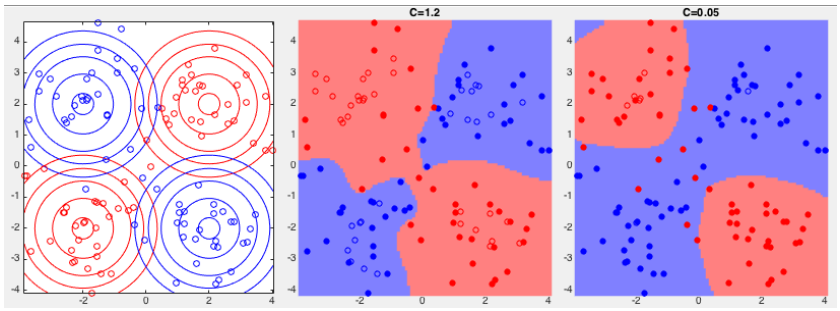


Figure 14.8 Classification of XOR Dataset by SVM Based on SMO

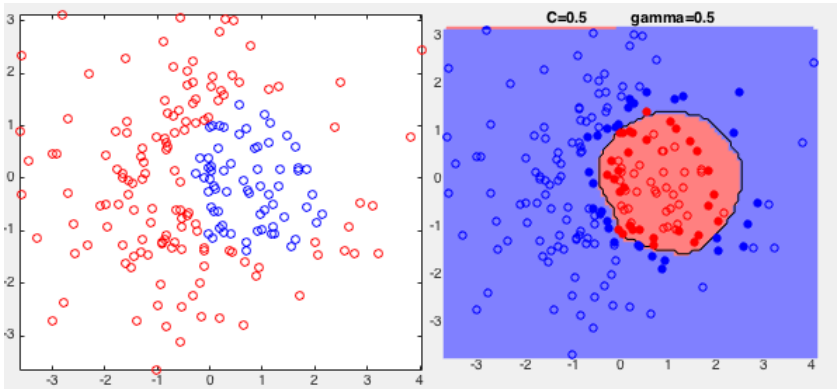


Figure 14.9 Example of Classification by SMO

One the other hand, when $C = 0.05$ is small, the decision boundary is determined by a greater number of support vectors, better reflecting the global distribution of the data points. However, as the local points close to the boundary are not relatively de-emphosized, some miss-classification occurs, some nine red dots are classified into the blue region incorrectly.

Two additional examples of binary classification based on SMO are shown in Figs. 14.9 and 14.10

14.5 Multiclass Classification

The SVM algorithm is inherently a binary classifier, but it can be generalized to multiclass problems as well. In this section, we consider a general K-class classifier based on a training set $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, of which each sample $\mathbf{x}_n$ is labeled by $y_n \in \{1, \cdots, K\}$ to indicate $\mathbf{x}_n \in C_{y_n}$.

We first consider two straight forward and imperial methods for multiclass classification based directly on binary SVM.

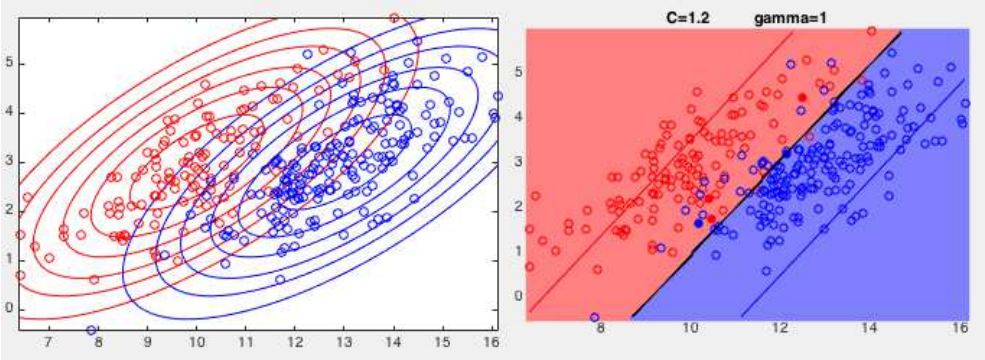


Figure 14.10 Example of Classification by SMO

- One-Vs-One (1V1)

  Any unlabeled $\mathbf{x}$ is classified to a class which receives the maximum votes out of the $K(K-1)/2$ binary classifications between every pair of the K classes.

- One-Vs-Rest (1VR)

  This method converts a K-class problem ($K > 2$) into K binary problems. First, we regroup the K classes $C_1, \cdots, C_K$ into two classes $C_+ = C_k$ and $C_- = \{C_l | l = 1, \cdots, K,\ l \neq k\}$ and find the corresponding decision function $f_k(\mathbf{x})$ representing quantitatively to what extent a given $\mathbf{x}$ belongs to class $C_+ = C_k$ (if $f_k(\mathbf{x}) > 0$), instead of C_- containing all remaining $K-1$ classes (if $f_k(\mathbf{x}) < 0$). After this process has been carried out for all $k = 1, \cdots, K$, the $\mathbf{x}$ can be classified as below:

$$\text{if} \quad f_l(\mathbf{x}) = \max\{f_1(\mathbf{x}), \cdots, f_K(\mathbf{x})\}, \quad \text{then} \quad \mathbf{x} \in C_l \tag{14.82}$$

We now consider another method for multiclass classification by directly generalizing the binary SVM so that all K classes can be separated by the decision boundaries is such a way that the margin between any two of the K classes is maximized. First, as in the binary case, we define a linear function for each of the K classes:

$$f_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + b_k \quad (k = 1, \cdots, K) \tag{14.83}$$

Once the parameters $\mathbf{w}_k$ and b_k are determined during training, any unlabeled point $\mathbf{x}$ can be classified as below:

$$\text{if} \ \ f_l(\mathbf{x}) = \max_k\{f_k(\mathbf{x}) \ \ (k = 1, \cdots, K)\}, \quad \text{then} \ \ \mathbf{x} \in C_l \tag{14.84}$$

Fig. 14.11 illustrates the classification of $K = 3$ classes in $d = 2$ dimensional feature space. Here each straight line H_k (plane or hyperplane if $d \geq 3$) is determined by equation $f_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + b_k = 0$, $(k = 1, \cdots, K)$, with normal direction in $\mathbf{w}_k$ and distance $d(H_k, \mathbf{0}) = |b_k|/||\mathbf{w}_k||$ to the origin (the same as in Eq. (14.4)).

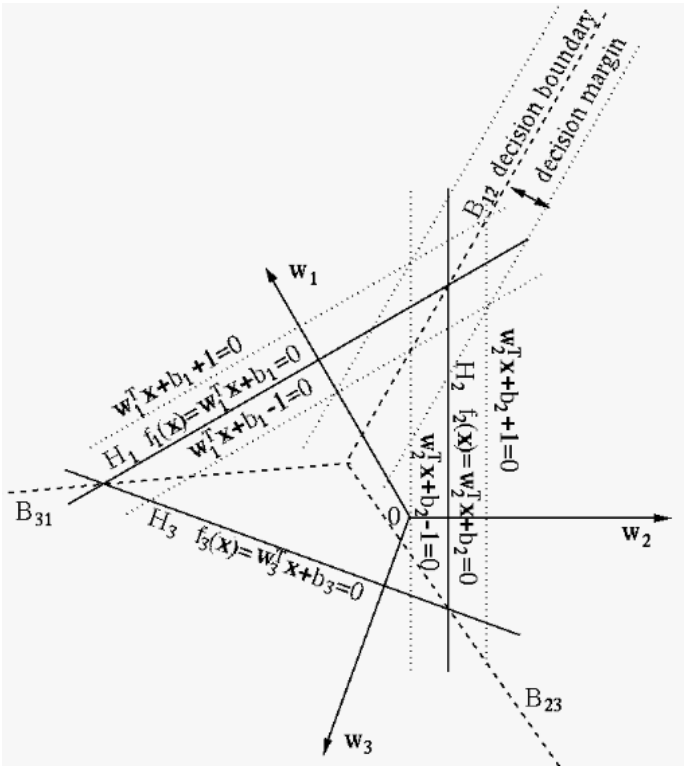


Figure 14.11 Multiclass SMV

According to the classification rule in Eq. (14.84), the 2-D feature space is partitioned into three regions each for one of the $K = 3$ classes by the decision boundaries $B_{12},\ B_{23},\ B_{31}$, where B_{ij} between classes C_i and C_j is composed of all points $\mathbf{x} \in B_{ij}$ satisfying $f_i(\mathbf{x}) = f_j(\mathbf{x})$. Like in binary SVM case, we further define for each of the K classes two additional straight lines (planes or hyperplanes if $d \geq 3$) in parallel with and on either side of H_k, denoted by $H_{k\pm}$ and represented by equations $f_k(\mathbf{x}) = \mathbf{w}_k^T\mathbf{x} + b_k \pm c_k$. This distance between these straight lines and H_k is (the same as in Eq. (14.9)):

$$d(H_{k\pm}, H_k) = \frac{|c_k|}{||\mathbf{w}_k||} \tag{14.85}$$

Also as shown in the figure, the intersections of these parallel lines (two for each of class) further determine two straight lines parallel to the decision boundary B_{12}, defining the decision margin between the support vectors of C_1 and C_2. It can be seen that this decision margin is related to both $|c_1|/||\mathbf{w}_1||$ and $|c_2|/||\mathbf{w}_2||$, which are maximized if $||\mathbf{w}_1||$ and $||\mathbf{w}_2||$ are minimized.

Now the multiclass classification problem can be formulated as to find the parameters $\mathbf{w}_k$ and b_k for each of the K classes, so that $\sum_{k=1}^{K} ||\mathbf{w}_k||^2$ is minimized

and thereby the decision margins for all boundaries B_{ij} are maximized, under the constraint that every training sample $\mathbf{x}_n$ labeled by y_n is correctly classified with a decision margin of 1 (the distance between the support vectors on opposite sides of B_{ij} is 2):

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}\sum_{k=1}^{K} ||\mathbf{w}_k||^2 \\ \text{subject to:} \quad & \mathbf{w}_{y_n}^T \mathbf{x}_n + b_{y_n} \geq \mathbf{w}_k^T \mathbf{x}_n + b_k + 2 \quad (k = 1, \cdots, K,\ k \neq y_n) \end{aligned} \tag{14.86}$$

Similar to Eq. (14.37) in soft margin SVM, the constraints of this optimization problem can be relaxed by including in each constraint an extra error term $\xi_{nk} \geq 0$, which also needs to be minimized. Now the problem can be reformulated as

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}\sum_{k=1}^{K} ||\mathbf{w}_k||^2 + C\sum_{k=1}^{K}\sum_{n=1}^{N} \xi_{nk} \\ \text{subject to:} \quad & \mathbf{w}_{y_n}^T \mathbf{x}_n + b_{y_n} \geq \mathbf{w}_k^T \mathbf{x}_n + b_k + 2 - \xi_{nk} \\ & \xi_{nk} \geq 0 \\ & (n = 1, \cdots, N, \qquad k = 1, \cdots, K,\ k \neq y_n) \end{aligned} \tag{14.87}$$

Once this constrained optimization problem is solved we get $\mathbf{w}_k$ and b_k for all $k = 1, \cdots, K$, and we can find $f_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + b_k$ for any unlabeled sample $\mathbf{x}$ and classify it based on Eq. (14.84).

Similar to the binary SVM, we approach this optimization problem by considering the dual problem. We first construct the Lagrangian, the primal function, of this constrained optimization problem:

$$\begin{aligned} L_p(\mathbf{W}, \mathbf{b}, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = & \frac{1}{2}\sum_{k} ||\mathbf{w}_k||^2 + C\sum_{k}\sum_{n} \xi_{nk} \\ & - \sum_{k}\sum_{n} \alpha_{nk} \left[(\mathbf{w}_{y_n} - \mathbf{w}_k)^T \mathbf{x}_n + b_{y_n} - b_k - 2 + \xi_{nk} \right] \\ & - \sum_{k}\sum_{n} \beta_{nk} \xi_{nk} \end{aligned} \tag{14.88}$$

where $\mathbf{W}$, $\mathbf{b}$, $\boldsymbol{\xi}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ are matrices and vectors containing the corresponding parameters $\mathbf{w}_k$, b_k, ξ_{nk}, α_{nk} and β_{nk} for all $k = 1, \cdots, K$ and $n = 1, \cdots, N$. For the special case $k = y_n$ not included in the constraints, $\alpha_{ny_n} = \beta_{ny_n} = 0$, and the constraint becomes an equality $0 = 2 - \xi_{ny_n}$, i.e., $\xi_{ny_n} = 2$.

The KKT conditions for the Lagrangian in Eq. (14.88) are listed below (similar to Eqs. (14.40) through (14.45) for the Lagrangian in Eq. (14.39)):

- Stationarity:
  –

$$\frac{\partial}{\partial \mathbf{w}_k} L_p(\mathbf{w}, b, \xi, \alpha, \beta) = \mathbf{w}_k + \sum_{n} \alpha_{nk} \mathbf{x}_n - \sum_{n} c_{nk} A_n \mathbf{x}_n = 0 \tag{14.89}$$

i.e.,

$$\mathbf{w}_k = \sum_n (c_{nk} A_n - \alpha_{nk}) \mathbf{x}_n \tag{14.90}$$

where we have defined

$$c_{nk} = \begin{cases} 1 & \text{if} \quad k = y_n \\ 0 & \text{else} \end{cases}, \quad \text{and} \quad A_n = \sum_k \alpha_{nk} \tag{14.91}$$

–

$$\frac{\partial}{\partial b_k} L_p(\mathbf{w}, b, \xi, \alpha, \beta) = -\sum_n c_{nk} A_n + \sum_n \alpha_{nk} = 0 \tag{14.92}$$

–

$$\frac{\partial}{\partial \xi_{nk}} L_p(\mathbf{w}, b, \xi, \alpha, \beta) = C - \alpha_{nk} - \beta_{nk} = 0 \tag{14.93}$$

- Primal and dual feasibilities:

$$\begin{aligned} &(\mathbf{w}_{y_n} - \mathbf{w}_k)^T \mathbf{x}_n + b_{y_n} - b_k - 2 + \xi_{nk} \geq 0 \\ &\alpha_{nk} \geq 0, \quad \beta_{nk} \geq 0, \quad \xi_{nk} \geq 0 \\ &\qquad (n = 1, \cdots, N, \quad k = 1, \cdots, K, \ k \neq y_n) \end{aligned} \tag{14.94}$$

- Complementarity:

$$\alpha_{nk}[(\mathbf{w}_{y_n} - \mathbf{w}_k)^T \mathbf{x}_n + b_{y_n} - b_k - 2 + \xi_{nk}] = 0 \tag{14.95}$$

These KKT conditions need to be satisfied by the solution of the optimization problem, including $\mathbf{w}_k^*$, b_k^*, and ξ_{nk}^* that need to satisfy the stationarity of the KKT conditions. We can therefore substitute Eq. (14.90) into Eq. (14.88) and get

$$\begin{aligned} L_p(\mathbf{W}, \mathbf{b}, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) &= \frac{1}{2} \sum_k \left(\sum_m (c_{mk} A_m - \alpha_{mk}) \right) \left(\sum_n (c_{nk} A_n - \alpha_{nk}) \right) \mathbf{x}_m^T \mathbf{x}_n \\ &- \sum_k \sum_n \alpha_{nk} \left[\sum_m (c_{my_n} A_m - \alpha_{my_n}) \mathbf{x}_m^T \mathbf{x}_n - \sum_m (c_{mk} A_m - \alpha_{mk}) \mathbf{x}_m^T \mathbf{x}_n + b_{y_n} - b_k - 2 \right] \\ &- \sum_k \sum_n \alpha_{nk} \xi_{nk} + C \sum_k \sum_n \xi_{nk} - \sum_k \sum_n \beta_{nk} \xi_{nk} \end{aligned} \tag{14.96}$$

This expression can be significantly simplified. First, we note that the last three terms cancel out due to Eq. (14.93). Second, the two terms containing b_{y_n} and b_k can be written as

$$\begin{aligned} \sum_k \sum_n \alpha_{nk} b_{y_n} &= \sum_k \sum_n c_{nk} \alpha_{nk} b_k = \sum_k b_k \sum_n A_n c_{nk} \\ \sum_k \sum_n \alpha_{nk} b_k &= \sum_k b_k \sum_n \alpha_{nk} \end{aligned} \tag{14.97}$$

and they also cancel out due to Eq. (14.92).

Now we can rewrite L_p as a function of only the variable $\boldsymbol{\alpha}$:

$$
\begin{aligned}
L_d(\boldsymbol{\alpha}) &\overset{1}{=} 2\sum_k\sum_n \alpha_{nk} \\
&+ \sum_k\sum_m\sum_n \left[\frac{1}{2}c_{mk}c_{nk}A_mA_n - c_{mk}\alpha_{nk}A_m + \frac{1}{2}\alpha_{mk}\alpha_{nk} - c_{my_n}\alpha_{nk}A_m + \alpha_{nk}\alpha_{my_n} + c_{mk}\alpha_{nk}A_m - \alpha_{nk}\alpha_{mk}\right] \\
&= 2\sum_k\sum_n \alpha_{nk} + \sum_k\sum_m\sum_n \left[\frac{1}{2}c_{mk}c_{nk}A_mA_n - \frac{1}{2}\alpha_{mk}\alpha_{nk} - c_{my_n}\alpha_{nk}A_m + \alpha_{nk}\alpha_{my_n}\right]\mathbf{x}_m^T\mathbf{x}_n \\
&\overset{2}{=} 2\sum_k\sum_n \alpha_{nk} + \sum_k\sum_m\sum_n \left[-\frac{1}{2}c_{my_n}A_mA_n - \frac{1}{2}\alpha_{mk}\alpha_{nk} + \alpha_{nk}\alpha_{my_n}\right]\mathbf{x}_m^T\mathbf{x}_n
\end{aligned}
$$

where the labeled equal signs are due to the following:

1. The two cross terms of the second part can be combined as:

$$\sum_m\sum_n\sum_k c_{mk}\alpha_{nk}A_m = \sum_m\sum_n\sum_k c_{nk}\alpha_{mk}A_n \tag{14.99}$$

2. As $c_{my_n} = 1$ if $y_m = y_n$ but zero otherwise, we have

$$\sum_k c_{mk}c_{nk} = c_{my_n} = \begin{cases} 1 & y_m = y_n \\ 0 & \text{otherwise} \end{cases} \tag{14.100}$$

and the first term of the second part can be written as $A_mA_nc_{my_n}/2$. Also, due to Eq. (14.92), the third term can be written as $-c_{my_n}c_{nk}A_nA_m = -c_{my_n}A_nA_m$ (when summed over k) and combined with the first one.

Eq. (14.98) is the quadratic dual function that needs to be maximized with respect to $\boldsymbol{\alpha}$ for it to be the tightest lower bound, subject to the linear constraint imposed by Eq. (14.92). We therefore have the following dual problem:

$$
\begin{aligned}
\text{maximize:} \quad & L_d(\boldsymbol{\alpha}) \\
\text{subject to:} \quad & \sum_n A_nc_{nk} = \sum_n \alpha_{nk} \\
& 0 \le \alpha_{nk} \le C, \quad \alpha_{ny_n} = 0 \\
& (n = 1, \cdots, N, \qquad k = 1, \cdots, K,\ k \ne y_n)
\end{aligned} \tag{14.101}
$$

This is a quadratic programming (QP) problem containing NK variables α_{nk}, $(n = 1, \cdots,\ k = 1, \cdots, K)$, solving which we get the optimal $\boldsymbol{\alpha}^*$.

We can further get all K optimal weight vectors $\mathbf{w}_k^*$ by Eq. (14.90), and $\mathbf{b}^*$ based on the complementarity of the KKT conditions, by solving the simultaneous equations in Eq. (14.95) (similar to how we get b^* in binary case in Eq. (14.30). Having found both optimal parameters $\mathbf{w}_k^*$ and b_k^*, we can classify any unlabeled pattern $\mathbf{x}$ by Eq. (14.84).

We note that ass the data points in Eq. (14.98) appear only in the form of an inner product, kernel method can be used to map the problem into a high dimensional space where the data points are better separable.

The algorithm discussed here can be considered the most classic multiclass

SVM algorithm as it is the first SVM algorithm to address multiclass problems by directly generalizing the binary SVM. Due to some of its drawback such as the size of variable $\boldsymbol{\alpha}$ in the dual problem (NK versus N in Eq. (14.27) in the binary case), various algorithms are developed later for improvement in computational complexity.

14.6 Kernelized Bayes classifier

The naive Bayes classification is based on a quadratic decision function and is therefore unable to classify data that are not quadratically separable. However, as shown below, the Bayes method can be reformulated so that all data points appear in the form of an inner product, and the kernel method can be used to map the original space into a higher dimensional space in which all groups can be separated even though they are not quadratically separable in the original space.

Consider a binary Bayes classifier by which any sample $\mathbf{x}$ is classified into either of the two classes C_+ and C_- depending on whether $\mathbf{x}$ is on the positive or negative side of the quadratic decision surface (Eq. (13.23)):

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{W}\mathbf{x} + \mathbf{w}^T\mathbf{x} + w \begin{cases} > 0, & \mathbf{x} \in C_+ \\ < 0, & \mathbf{x} \in C_- \end{cases} \tag{14.102}$$

As show in Eq. (13.22), here

$$\begin{aligned} \mathbf{W} &= -\frac{1}{2}(\boldsymbol{\Sigma}_+^{-1} - \boldsymbol{\Sigma}_-^{-1}) \\ \mathbf{w} &= \boldsymbol{\Sigma}_+^{-1}\mathbf{m}_+ - \boldsymbol{\Sigma}_-^{-1}\mathbf{m}_- \\ w &= -\frac{1}{2}(\mathbf{m}_+^T\boldsymbol{\Sigma}_+^{-1}\mathbf{m}_+ - \mathbf{m}_-^T\boldsymbol{\Sigma}_-^{-1}\mathbf{m}_-) - \frac{1}{2}\log\frac{|\boldsymbol{\Sigma}_+|}{|\boldsymbol{\Sigma}_-|} + \log\frac{P(C_+)}{P(C_-)} \end{aligned} \tag{14.103}$$

where $\boldsymbol{\Sigma}_+$ and $\boldsymbol{\Sigma}_-$ are respectively the covariance matrices of the samples in C_+ and C_-,

$$\mathbf{m}_+ = \frac{1}{N_+}\sum_{x\in C_+}\mathbf{x}, \qquad \mathbf{m}_- = \frac{1}{N_-}\sum_{x\in C_-}\mathbf{x} \tag{14.104}$$

are their mean vectors, and $P_+ = N_+/N$ and $P_- = N_-/N$ are their prior probabilities. Specially if $\boldsymbol{\Sigma}_+ = \boldsymbol{\Sigma}_- = \boldsymbol{\Sigma}$ and therefore $\mathbf{W} = \mathbf{0}$, the quadratic decision surface becomes a linear decision plane described by

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w = \left[\boldsymbol{\Sigma}^{-1}(\mathbf{m}_+ - \mathbf{m}_-)\right]^T\mathbf{x} + w = 0 \tag{14.105}$$

in the same form as the decision equation for the support vector machine. An unlabeled sample $\mathbf{x}$ is classified into either of the two classes depending on on which side of a threshold $-w$ its projection onto the normal direction $\mathbf{w}$ of the decision plane lies.

As matrices $\mathbf{W}$, $\mathbf{\Sigma}_+^{-1}$, and $\mathbf{\Sigma}_-^{-1}$ are all symmetric, they can be written in the following eigen-decomposition form:

$$\mathbf{W} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T = \mathbf{U}\mathbf{U}^T, \quad \mathbf{\Sigma}_+^{-1} = \mathbf{V}_+\mathbf{\Lambda}_+\mathbf{V}_+^T = \mathbf{U}_+\mathbf{U}_+^T, \quad \mathbf{\Sigma}_-^{-1} = \mathbf{V}_-\mathbf{\Lambda}_-\mathbf{V}_-^T = \mathbf{U}_-\mathbf{U}_-^T \tag{14.106}$$

where $\mathbf{U} = \mathbf{V}\mathbf{\Lambda}^{1/2}$, $\mathbf{U}_+ = \mathbf{V}_+\mathbf{\Lambda}_+^{1/2}$ and $\mathbf{U}_- = \mathbf{V}_-\mathbf{\Lambda}_-^{1/2}$. We can write vector $\mathbf{w}$ as:

$$\mathbf{w} = \mathbf{\Sigma}_+^{-1}\mathbf{m}_+ - \mathbf{\Sigma}_-^{-1}\mathbf{m}_- = \mathbf{U}_+\mathbf{U}_+^T\frac{1}{N_+}\sum_{\mathbf{x}_+\in C_+}\mathbf{x}_+ - \mathbf{U}_-\mathbf{U}_-^T\frac{1}{N_-}\sum_{\mathbf{x}_-\in C_-}\mathbf{x}_- \tag{14.107}$$

Any unlabeled sample $\mathbf{x}$ can now be classified into either of the two classes based on its decision function:

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{x}^T\mathbf{W}\mathbf{x} + \mathbf{w}^T\mathbf{x} + w \\ &= \mathbf{x}^T\mathbf{U}\mathbf{U}^T\mathbf{x} + \left(\frac{1}{N_+}\sum_{\mathbf{x}_+}\mathbf{x}_+^T\mathbf{U}_+\mathbf{U}_+^T\right)\mathbf{x} - \left(\frac{1}{N_-}\sum_{\mathbf{x}_-}\mathbf{x}_-^T\mathbf{U}_-\mathbf{U}_-^T\right)\mathbf{x} + w \\ &= \mathbf{z}^T\mathbf{z} + \frac{1}{N_+}\sum_{\mathbf{z}_{++}}\left(\mathbf{z}_{++}^T\mathbf{z}_+\right) - \frac{1}{N_-}\sum_{\mathbf{z}_{--}}\left(\mathbf{z}_{--}^T\mathbf{z}_-\right) + w \end{aligned} \tag{14.108}$$

where

$$\begin{cases} \mathbf{z}_{++} = \mathbf{U}_+^T\mathbf{x}_+ \ \ (\mathbf{x}_+ \in C_+) \\ \mathbf{z}_{--} = \mathbf{U}_-^T\mathbf{x}_- \ \ (\mathbf{x}_- \in C_-) \end{cases}, \quad \begin{cases} \mathbf{z} = \mathbf{U}^T\mathbf{x} \\ \mathbf{z}_+ = \mathbf{U}_+^T\mathbf{x} \\ \mathbf{z}_- = \mathbf{U}_-^T\mathbf{x} \end{cases} \tag{14.109}$$

As all data points appear in the form of an inner product, the kernel method can be used by replacing all inner products in the decision function by the corresponding kernel functions:

$$f(\mathbf{x}) = K(\mathbf{z},\mathbf{z}) + \frac{1}{N_+}\sum_{\mathbf{z}_{++}}K(\mathbf{z}_{++},\mathbf{z}_+) - \frac{1}{N_-}\sum_{\mathbf{z}_{--}}K(\mathbf{z}_{--},\mathbf{z}_-) + b = p(\mathbf{x}) + b \tag{14.110}$$

This is the decision function in the new higher dimensional space, where $p(\mathbf{x})$ is defined as a function composed of all terms in $f(\mathbf{x})$ except the last offset term b, which is to replace w in the original space. To find this b, we first map all training samples into a 1-D space $x_n = p(\mathbf{x}_n)$, $(n = 1, \cdots, N)$ and sort them together with their corresponding labelings $y_1, \cdots, y_N$, and then search through all $N-1$ possible ways to partition them into two groups indexed respectively by $1, \cdots, k$ and $k+1, \cdots, N$ to find the optimal k corresponding to the maximum labeling consistency measured by

$$\left|\sum_{n=1}^{k} y_n\right| + \left|\sum_{n=k+1}^{N} y_n\right|, \qquad (k = 1, \cdots, N-1) \tag{14.111}$$

The middle point $(x_k + x_{k+1})/2$ between x_k and x_{k+1} is used as the optimal threshold to separate the training samples into two classes in the 1-D space, i.e.,

the offset is $b = -(x_k + x_{k+1})/2$. Now the unlabeled point $\mathbf{x}$ can be classified into either of the two classes C_+ and C_-:

$$p(\mathbf{x}) + b \begin{cases} > 0, & \mathbf{x} \in C_+ \\ < 0, & \mathbf{x} \in C_- \end{cases} \tag{14.112}$$

In general, when all data points are kernel mapped to a higher dimensional space, the two classes can be more easily separated, even by a hyperplane based on the linear part of the decision function without the second order term. This allows the assumption that the two classes have the same covariance matrix so that $\mathbf{W} = -(\mathbf{\Sigma}_+ - \mathbf{\Sigma}_-)/2 = \mathbf{0}$ and the second order term is dropped. This is the justification for the following two special cases:

- If we approximate both $\mathbf{\Sigma}_+$ and $\mathbf{\Sigma}_-$ by their average $\mathbf{\Sigma} = (\mathbf{\Sigma}_+ + \mathbf{\Sigma}_-)/2$, then $\mathbf{W} = \mathbf{0}$ and the decision function of any $\mathbf{x}$ becomes

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} + w = \left[\mathbf{\Sigma}^{-1}(\mathbf{m}_+ - \mathbf{m}_-)\right]^T \mathbf{x} + w \\ &= \left[\mathbf{U}\mathbf{U}^T \left(\frac{1}{N_+}\sum_{\mathbf{x}_+}\mathbf{x}_+ - \frac{1}{N_-}\sum_{\mathbf{x}_-}\mathbf{x}_-\right)\right]^T \mathbf{x} + w \\ &= \frac{1}{N_+}\sum_{\mathbf{x}_+}\mathbf{x}_+^T\mathbf{U}\mathbf{U}^T\mathbf{x} - \frac{1}{N_-}\sum_{\mathbf{x}_-}\mathbf{x}_-^T\mathbf{U}\mathbf{U}^T\mathbf{x} + w \\ &= \frac{1}{N_+}\sum_{\mathbf{z}_+}\mathbf{z}_+^T\mathbf{z} - \frac{1}{N_-}\sum_{\mathbf{z}_-}\mathbf{z}_-^T\mathbf{z} + w \end{aligned} \tag{14.113}$$

  where $\mathbf{U}\mathbf{U}^T = \mathbf{\Sigma}^{-1}$, $\mathbf{z}_+ = \mathbf{U}^T\mathbf{x}_+$, $\mathbf{z}_- = \mathbf{U}^T\mathbf{x}_-$, and $\mathbf{z} = \mathbf{U}^T\mathbf{x}$. This decision function can be converted to the following if the kernel method is used:

$$f(\mathbf{x}) = \frac{1}{N_+}\sum_{\mathbf{z}_+} K(\mathbf{z}_+, \mathbf{z}) - \frac{1}{N_-}\sum_{\mathbf{z}_-} K(\mathbf{z}_-, \mathbf{z}) + b \tag{14.114}$$

- More specially, if $\mathbf{\Sigma}_+ = \mathbf{\Sigma}_- = \mathbf{I}$, then the decision function above becomes

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} + w = (\mathbf{m}_+ - \mathbf{m}_-)^T \mathbf{x} + w \\ &= \frac{1}{N_+}\sum_{\mathbf{x}_+}\mathbf{x}_+^T\mathbf{x} - \frac{1}{N_-}\sum_{\mathbf{x}_-}\mathbf{x}_-^T\mathbf{x} + w \end{aligned} \tag{14.115}$$

  which can be converted to the following if the kernel method is used:

$$f(\mathbf{x}) = \frac{1}{N_+}\sum_{\mathbf{x}_+} K(\mathbf{x}_+, \mathbf{x}) - \frac{1}{N_-}\sum_{\mathbf{x}_-} K(\mathbf{x}_-, \mathbf{x}) + b \tag{14.116}$$

Note that if the kernel method is used to replace an inner product by a kernel function $K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T\phi(\mathbf{x}')$, we need to map all data points to $\phi(\mathbf{x}_n)$, $(n = 1, \cdots, N)$ in the higher dimensional space, instead of only mapping their means

$\phi(\mathbf{m}_+)$ and $\phi(\mathbf{m}_-)$, because the mapping of a sum is not equal to the sum of the mapped points if the kernel is not linear:

$$\phi(\mathbf{m}_+ - \mathbf{m}_-) \neq \phi(\mathbf{m}_+) - \phi(\mathbf{m}_-), \qquad \phi(\mathbf{m}_\pm) = \phi\left(\frac{1}{n_\pm}\sum_{\mathbf{x}\in C_\pm}\mathbf{x}\right) \neq \frac{1}{n_\pm}\sum_{\mathbf{x}\in C_\pm}\phi(\mathbf{x}) \tag{14.117}$$

The three cases above are summarized below:

- Type I, Eq. (14.116)

$$f(\mathbf{x}) = \frac{1}{N_+}\sum_{\mathbf{x}_+} K(\mathbf{x}_+, \mathbf{x}) - \frac{1}{N_-}\sum_{\mathbf{x}_-} K(\mathbf{x}_-, \mathbf{x}) + b \tag{14.118}$$

- Type II, Eq. (14.113)

$$f(\mathbf{x}) = \frac{1}{N_+}\sum_{\mathbf{z}_+} K(\mathbf{z}_+, \mathbf{z}) - \frac{1}{N_-}\sum_{\mathbf{z}_-} K(\mathbf{z}_-, \mathbf{z}) + b \tag{14.119}$$

- Type III, Eq. (14.110)

$$f(\mathbf{x}) = \mathbf{z}^T\mathbf{z} + \frac{1}{N_+}\sum_{\mathbf{z}_{++}} K(\mathbf{z}_{++}, \mathbf{z}_+) - \frac{1}{N_-}\sum_{\mathbf{z}_{--}} K(\mathbf{z}_{--}, \mathbf{z}_-) + b \tag{14.120}$$

The Matlab code for the essential part of the algorithm is listed below. Given `X` and `y` for the data array composed of N training vectors $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ each labeled by the corresponding component $\mathbf{y} = [y_1, \cdots, y_N]$, the code carries out the training and then classifies any unlabeled data point into either of the two classes. Parameter `type` selects any one of the three different versions of the algorithm, and the function `K(X,x)` returns a 1-D array containing all N kernel functions $K(\mathbf{x}_i, \mathbf{x})$, $(i = 1, \cdots, N)$ of the column vectors in $\mathbf{X}$ and vector $\mathbf{x}$.

```
X=getData;
[m n]=size(X);
X0=X(:,find(y>0));   n0=size(X0,2); % separate C+ and C-
X1=X(:,find(y<0));   n1=size(X1,2);
m0=mean(X0,2);    C0=cov(X0');      % mean and covariance
m1=mean(X1,2);    C1=cov(X1');
if type==1
    for i=1:n
        x(i)=sum(K(X0,X(:,i)))/n0-sum(K(X1,X(:,i)))/n1;
    end
elseif type==2
    C=inv(C0+C1);   [V D]=eig(C);   U=(V*D^(1/2))';
    Z=U*X;  Z0=U*X0;   Z1=U*X1;
    for i=1:n
        x(i)=sum(K(Z0,Z(:,i)))/n0-sum(K(Z1,Z(:,i)))/n1;
    end
```

```
elseif type==3
    C0=inv(C0);   C1=inv(C1);   W=-(C0-C1)/2;
    [V D]=eig(W);     U=(V*D^(1/2)).';    Z=U*X;
    [V0 D0]=eig(C0);  U0=(V0*D0^(1/2))';  Z0=U0*X;  Z00=U0*X0;
    [V1 D1]=eig(C1);  U1=(V1*D1^(1/2))';  Z1=U1*X;  Z11=U1*X1;
    for i=1:n
        x(i)=K(Z(:,i),Z(:,i))+sum(K(Z00,Z0(:,i)))/n0-sum(K(Z11,Z1(:,i)))/n1;
    end
end
[x I]=sort(x);   y=y(I);    % sort 1-D data and their labelings
smax=0;
for i=1:n-1                 % find optimal threshold value b
    s=abs(sum(y(1:i)))+abs(sum(y(i+1:n)));
    if s>smax
        smax=s;  b=-(x(i)+x(i+1))/2;
    end
end
```

Note that $\mathbf{W} = -(\mathbf{\Sigma}_0^{-1} - \mathbf{\Sigma}_1^{-1})/2$ may be either positive or negative definite, and its eigenvalue matrix $\mathbf{D}$ may contain negative values and $\mathbf{D}^{1/2}$ may contain complex values. Given any unlabeled data point $\mathbf{x}$, the code below is carried out

```
if type==1
    y(i)=sum(K(X0,x))/n0-sum(K(X1,x))/n1+b;
elseif type==2
    Z=U*x;
    y(i)=sum(K(Z0,z))/n0-sum(K(Z1,z))/n1+b;
elseif type==3
    z=U*x;    z0=U0*x;   z1=U1*x;
    y(i)=K(z,z)+sum(K(Z00,z0))/n0-sum(K(Z11,z1))/n1+b;
end
```

to classify $\mathbf{x}$ into either C_+ if $y > 0$, or C_- if $y < 0$.

In comparison with the SVM method, which requires solving a QP problem by certain iterative algorithm (either the interior point method or the SMO method), the kernel based Bayes method is closed-form and therefore extremely efficient computationally. Moreover, as shown in the examples below, this method is also highly effective as its classification results are comparable and often more accurate than those of the SVM method.

We now show a few examples to test all three types of the kernel based Bayes method based on a set of simulated 2-D data. Both linear kernel and RBF kernel $K(\mathbf{x}, \mathbf{y}) = \exp(-\gamma||\mathbf{x} - \mathbf{y}||)$. The value of the parameter γ used in the examples is 5, but it can be fine tuned in a wide range (e.g., $1 < \gamma < 9$ to make proper tradeoff between accuracy and avoiding overfitting. The performances of these

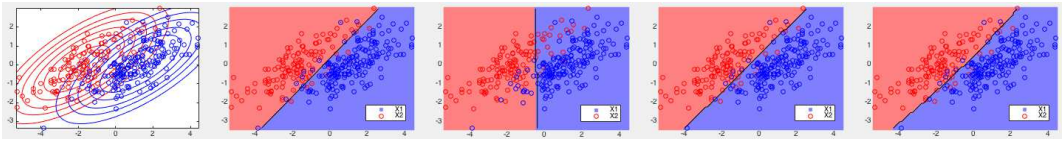


Figure 14.12 Test 1 of Kernelized Bayes Classifier

method are also compared with the SVM method implemented by the Matlab function:

```
fitcsvm(X',y,'Standardize',true,'KernelFunction','linear','KernelScale','auto'))
```

Example 14.5 Four different binary classifiers are applied to two 2-D datasets, with their results compared in terms of the correct rates listed below, and their partitionings of the 2-D space shown in Figs. 14.12 and 14.13.

	Kernel	Matlab SVM	Kernel Bayes I	Kernel Bayes II	Kernel Bayes III
Test 1	linear	93.0%	88.0%	94.0%	94.0%
Test 2	linear	73.0%	80.0%	79.5%	96.5%
Test 3	RBF	98.5%	100%	100%	100%

(14.121)

- Test 1: Two sets of data are generated based on the following mean vectors and covariance matrices:

$$\mathbf{m}_+ = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \mathbf{m}_- = \begin{bmatrix} +1 \\ 0 \end{bmatrix}, \quad \boldsymbol{\Sigma}_+ = \boldsymbol{\Sigma}_- = 3 \times \begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \tag{14.122}$$

  The linear kernel is used for all methods. The kernel Bayes methods II and III achieve the best result (94%), slightly better than that of the standard SVM (93%). But the kernel Bayes method I performs poorly (84.7%), as it is based only on the means of the two classes without taking into consideration their covariances representing the distribution of the data points.
- Test 2: Two sets of data not linearly separable are classified by the four methods all based on the linear kernel. The kernel Bayes III achieves the best result (96.5%), due to its quadratic term in the decision function, the kernel Bayes I and II perform much more poorly, but still slightly better than the SVM method.
- Test 3: The same dataset used in Test 2 is classified but now based on the RBF kernel. While the SVM performs performs reasonably well (98.5%), all three versions of the kernel Bayes method (with $\gamma = 5$) achieve the perfect result with all points of the two classes completely separated. However, the partitioning of the space by the kernel Bayes III is highly fragmented, indicating an obvious overfitting.

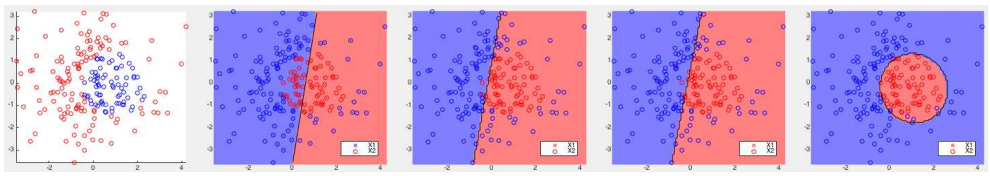


Figure 14.13 Test 2 of Kernelized Bayes Classifier

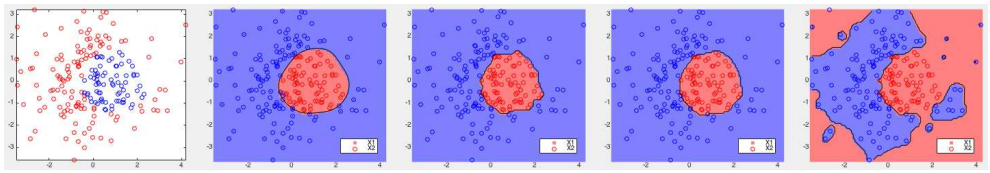


Figure 14.14 Test 3 of Kernelized Bayes Classifier

Example 14.6 Three datasets (XOR, cocentric rings, multi-clusters) are generated using the Matlab code at: `https://www.mathworks.com/help/stats/support-vector-machi`

The following four tests are carried out for the three versions of the kernel Bayes all based on a parameter $\gamma = 5$.

- Test 1: Exclusive OR dataset, linear kernel
  As the data are not linearly separable, all methods performed poorly except kernelized Bayes Type III with a second order term in the decision function. The results are shown in Fig. 14.15.
- Test 2: Exclusive OR dataset, RBF kernel
  All four methods performed very well, but all three variations of the kernelized Bayes method achieved higher correct rates than the SVM. The results are shown in Fig. 14.16.
- Test 3: Cocentric ring dataset, RBF kernel. The results are shown in Fig. 14.17.
- Test 4: Multi-cluster dataset, RBF kernel. The results are shown in Fig. 14.18.

The classification correct rates of the four tests are listed below:

	Kernel	Matlab SVM	Kernel Bayes I	Kernel Bayes II	Kernel Bayes III
Test 1	Linear	58.75%	60.0%	60.25%	97.0%
Test 2	RBF	97.75%	98.50%	98.50%	99.50%
Test 3	RBF	98.0%	100%	100%	100%
Test 4	RBF	96.0%	98.5%	95.5%	97.0%

Figs. through 14.18.

Example 14.7 In this example both the kernel Bayes and SVM methods (by Matlab function `fitcsvm`, see `https://www.mathworks.com/help/stats/fitcsvm.html`) are applied to the iris dataset. Their results are compared in Fig. 14.19. Here

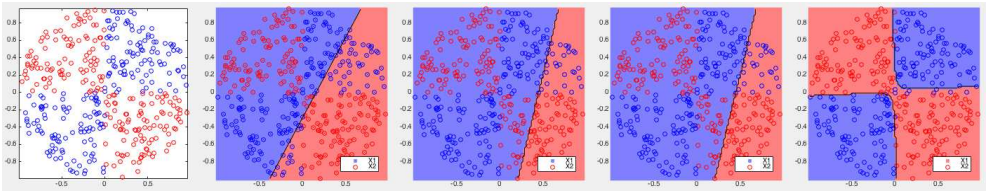


Figure 14.15 Example 14.6 test 1

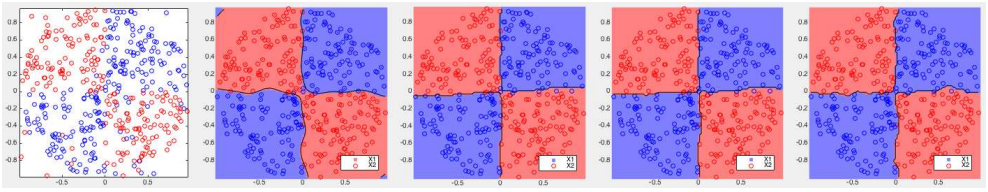


Figure 14.16 Example 14.6 test 2

only two (3rd and 4th) of the four features are used, with half of the samples used for training while the other half for testing. Note that the second class (setosa in green) and third class (versicolor in blue) not linearly separable can be better separated by the kernel Bayes method.

Problems

In this homework, you are to develop your own code to implement the binary and multiclass SVM algorithms. You are also encouraged to compare your results with those obtained by the built-in SVM functions in Matlab.

1. Develop code to implement the SVM algorithm using your own function for quadratic programming (QP) by the interior point algorithm (Section 3.7), and built-in QP function *quadprog* in Matlab, based on the radial basis function (RBF) kernel $K(\mathbf{x}_i, \mathbf{x}_j) = c \exp(-\gamma ||\mathbf{x}_i - \mathbf{x}_j||^2)$.

   Apply your code to carry out binary classification for each of the following 2-D datasets. Experiment with different values for the two hyper-parameters c

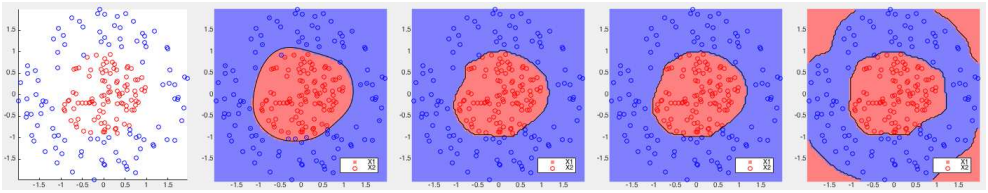


Figure 14.17 Example 14.6 test 3

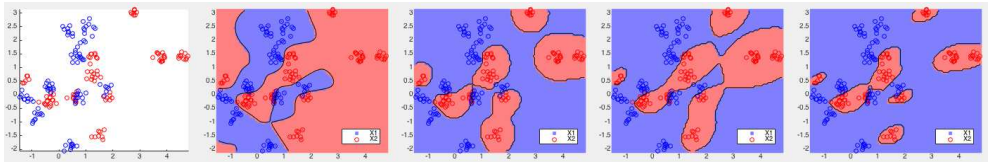


Figure 14.18 Example 14.6 test 4

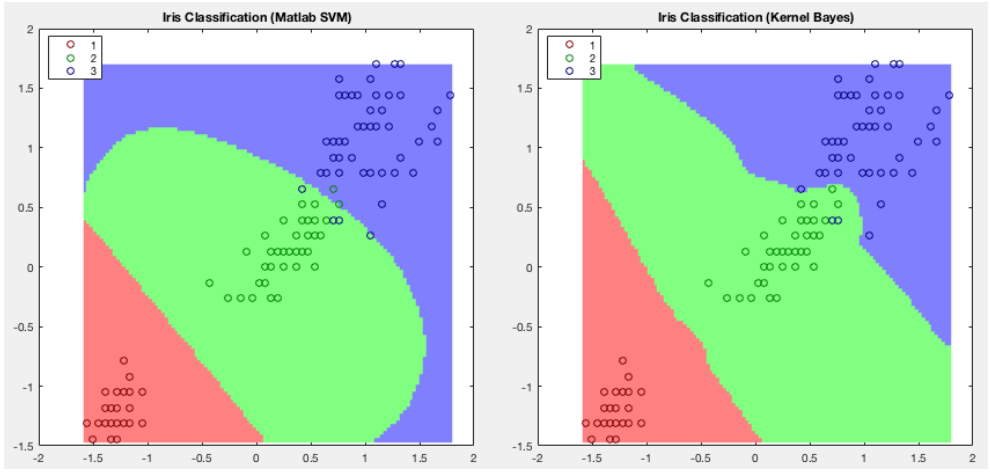


Figure 14.19 Comparison of kernel Bayes and SVM applied to iris data

and γ in the kernel function and check to see how they affect the classification result.

Visualize the partitioning of the 2-D feature space by classifying all points in the space (with proper resolution) into either of the two classes, same as the examples in

- /e176/programs/data60.txt
- /e176/programs/data61.txt
- /e176/programs/data62.txt
- /e176/programs/data63.txt
- /e176/programs/data64.txt

2. Carry out multiclass classification for both the iris and handwritten datasets, using your binary SVM code and the method of 1V1 or 1VR. Show your classification results in the form of confusion matrix as well as error rates.
3. Develop code for multiclass SVM and carry out classification for the iris and handwritten datasets. Show the classification results in the form of confusion matrix as well as error rates.

15 Clustering Analysis

All supervised classification algorithms previously considered are based on the assumed availability of some training data in which each of the data patterns $\mathbf{x}_n \in \mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N\}]$ is labeled by $y_n \in \mathbf{y} = [y_1, \cdots, y_n]^T$, i.e., the class they each belong is known. However, when such training set is not available, there is still the need to explore some potential structure of the data in the form of a set of unknown number K of clusters $\{C_1, \cdots, C_K\}$, each composed of a set of similar data points close to each other in the feature space. This can be accomplished by the unsupervised method of *clustering analysis*, based only on the given dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, without any additional prior knowledge, such as the labeling of the data samples. We will now consider a few algorithms for clustering analysis.

15.1 K-means clustering

As suggested by the name of K-means clustering, in this method a set of K mean vectors $\mathbf{m}_1, \cdots, \mathbf{m}_K$ are used to represent the clusters in the feature space, based on the assumption that there exist a set of K clusters in the dataset.

The K-means clustering algorithm can be formulated as an optimization problem to minimize an objective function

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} P_{nk} ||\mathbf{x}_n - \mathbf{m}_k||^2 \tag{15.1}$$

where

$$P_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_l ||\mathbf{x}_n - \mathbf{m}_l|| \\ 0 & \text{otherwise} \end{cases} \tag{15.2}$$

indicates that $\mathbf{x}_n$ is assigned to the kth cluster C_k if its distance to the mean $\mathbf{m}_k$ is minimum among all K such distances. To minimize the objective function J with respective to $\mathbf{m}_k$, we set its derivative with respect to each $\mathbf{m}_k$ to zero:

$$\frac{d}{d\mathbf{m}_k} J = 2 \sum_{n=1}^{N} P_{nk} ||\mathbf{x}_n - \mathbf{m}_k|| = \mathbf{0} \qquad k = 1, \cdots, K \tag{15.3}$$

and solve the resulting equation for $\mathbf{m}_k$:

$$\mathbf{m}_k = \frac{\sum_{n=1}^{N} P_{nk}\mathbf{x}_n}{\sum_{n=1}^{N} P_{nk}} = \frac{1}{N_k}\sum_{\mathbf{x}\in C_k}\mathbf{x} \quad (k = 1, \cdots, K) \tag{15.4}$$

where $N_k = \sum_{n=1}^{N} P_{nk}$ is the number of all samples assigned to cluster C_k, and the summation is over all samples assigned to C_k.

As P_{nk} depends on $\mathbf{m}_k$ which in turn also depends on P_{nk}, the steps above can be carried out iteratively, starting with some K randomly initialized mean vectors, which are then modified iteratively until convergence, as shown in the following steps:

1. Set $t = 0$, and initialize randomly the K mean vectors, such as any K samples randomly chosen from the dataset: $\mathbf{m}_1^{(0)}$, $\mathbf{m}_2^{(0)}, ..., \mathbf{m}_K^{(0)}$, set iteration index to zero $t = 1$;
2. Assign each sample $\mathbf{x}$ in the dataset to one of the K clusters according to its distance to the corresponding mean vector:

$$\text{if} \quad ||\mathbf{x} - \mathbf{m}_k^{(t)}||^2 = \min_{1\le l\le K} ||\mathbf{x} - \mathbf{m}_l^{(t)}||^2, \quad \text{then} \quad \mathbf{x} \in C_k^{(t)} \tag{15.5}$$

3. Update the mean vectors to get the new mean $\mathbf{m}_k^{(t+1)}$ so that the objective function J given above is minimized:

$$\mathbf{m}_k^{(t+1)} = \frac{1}{N_k}\sum_{\mathbf{x}\in C_k}\mathbf{x}, \qquad (k = 1, \cdots, K) \tag{15.6}$$

4. Terminate if the algorithm has converged:

$$\mathbf{m}_k^{(l+1)} = \mathbf{m}_k^{(t)} \quad (k = 1, \cdots, K) \tag{15.7}$$

   Otherwise, $t \leftarrow t + 1$, go back to Step 2.

The K-means clustering is simple and effective, but it has the main drawback that the ground truth number of clusters in the dataset is typically unknown, and it may be difficult to determine its vallue without any prior knowledge. One way to address this issue is to carry out the clustering process multiple times with different K values, and then determine the optimal K value by comparing the clustering results to see which best reflects the intrinsic structure of the dataset in terms of how the data points are distributed. How well the clustering result fits the data can be measured quantitatively by the separability of the clusters, such as the ratio between the intra and inter-cluster distances, or the ration between-cluster scatteredness and the total scatteredness of the entire dataset, i.e., $tr(\mathbf{S}_T^{-1}\mathbf{S}_B)$ in Eq. (9.24). The K value corresponding to the highest separability among all resulting clusters can be adopted as the optimal one. Such a quantitative measurement can also be used as a criterion other clustering algorithms, including the method of competitive learning network to be considered in Section 19.1.

The issue of unknown parameter K can also be addressed by modifying its

value during the iteration. For example, if the inter-cluster distance between two small clusters is smaller than a predetermined threshold, they can be merged into one, also, if the intra-cluster distance of a large cluster is greater than a predetermined threshold, it can be split into two. This method of modifying the clustering results by merging and splitting leads to the algorithm of *Iterative Self-Organizing Data Analysis Technique (ISODATA)*. However, this algorithm is highly empirical and it suffers from the drawback that it may be difficult to predetermine those thresholds without any prior knowledge or experience. It may therefore become an highly interactive process during which the threshold values themselves may need to be modified to better fit the data.

The Matlab code for the iteration loop of the K-means algorithm is listed below, where `Mold` and `Mnew` are respectively the K mean vectors before and after each modification. The iteration terminates when the mean vectors no longer change.

```
Mnew=X(:,randi(N,1,K));     % pick any K samples as initial means
er=inf;
while er>0                  % main iteration
    Mold=Mnew;
    Mnew=zeros(d,K);        % initialize new means
    Nk=zeros(1,K);
    for n=1:N               % for all N samples
        x=X(:,n);
        dmin=inf;
        for k=1:K           % for all K clusters
            d=norm(x-Mold(:,k));
            if d<dmin
                dmin=d;  j=k;
            end
        end
        Nk(j)=Nk(j)+1;
        Mnew(:,j)=Mnew(:,j)+x;
    end
    for k=1:K
        if Nk(k)>0
            Mnew(:,k)=Mnew(:,k)/Nk(k);
        end
    end
    er=norm(Mnew-Mold);     % terminate if means no longer change
end
```

Example 15.1 The K-means algorithm is applied to a simulated dataset in 3-D space with $C = 4$ clusters. The three panels in Fig. 15.1 show the results corresponding to $K = C-1$ (left), $K = C$ (middle), and $K = C+1$ (right). The

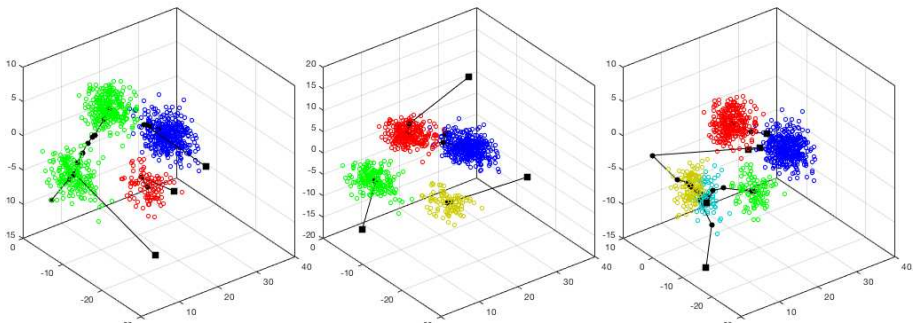


Figure 15.1 K-Means Clustering with $K = 3$ (left), $K = 4$ (middle), and $K = 5$ (right)

initial positions of the K means are marked by the black squares, while their subsequent positions through out the iteration are marked by smaller dots. The iteration terminates once the means have moved to the centers of the clusters and no longer change positions.

The clustering results corresponding to $K = 3, 4, 5$ can be evaluated by the separability $tr(\mathbf{S}_T^{-1}\mathbf{S}_B)$, the K intra-cluster distance $tr(\mathbf{\Sigma}_k)$ $(k = 1, \cdots, K)$ of the resulting clusters:

	$K = C - 1 = 3$			$K = C = 4$				$K = C + 1 = 5$				
Separability	1.76			2.56				2.58				
Intra-cluster distance	9.1	44.3	11.8	10.8	12.7	11.1	9.1	10.8	9.1	11.1	8.9	9.4

(15.8)

and the $K(K-1)/2$ inter-cluster (Bhattacharyya) distances (between any two of the K clusters):

$K = C - 1 = 3$		
	1	2
2	10.9	
3	21.9	184.7

$K = C = 4$			
	1	2	3
2	4.0		
3	5.1	1.6	
4	2.4	2.3	4.4

$K = C + 1 = 5$				
	1	2	3	4
2	4.0			
3	5.1	1.6		
4	7.0	4.3	0.3	
5	17.3	15.5	8.9	11.1

(15.9)

We see that when $K = 3 < C = 4$, the intra-cluster distance of the 2nd cluster is significantly greater than the other two, indicating the cluster may contain two smaller clusters. Also, when $K = 5 > C = 4$, the inter-cluster distance between clusters 3 and 4 is significantly smaller than others, indicating the two clusters are too close and can therefore be merged.

While the K-means method is simple and effective, it has the main shortcoming that the number of clusters K needs to be predetermined while it is typically unknown ahead of time without any prior knowledge. In this case, one could try different K values and compare the corresponding results in terms of the

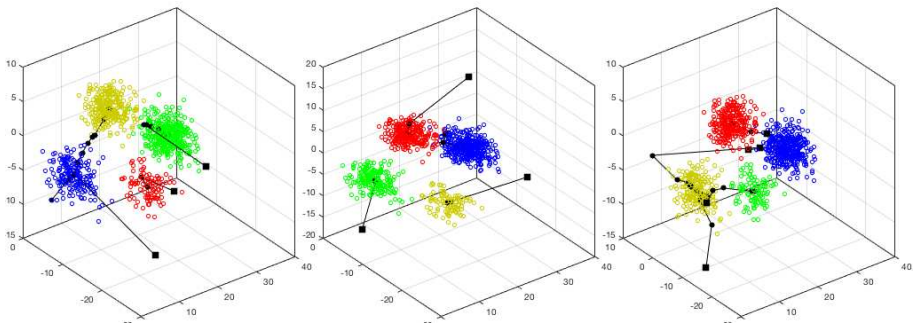


Figure 15.2 K-Means Clustering with Merge and Split

intra and inter cluster distances, as well as the separabilities of the resulting clusters, as shown in the example above. Moreover, if the intra-cluster distance of a cluster is too large indicating it may contain more than one cluster (e.g., $K = 3$ in the example), it can be split; on the other hand, if the inter-cluster distance between two clusters is too small (e.g., $K = 5$ in the example), the two clusters may belong to the same cluster and need to be merged. Following such merging and/or splitting, a few more iterations can be carried out to make sure the final clustering results are optimal. Fig. 15.2 shows the clustering results of the same dataset above but modified by merging and splitting. The left panel is for $K = 3$, but the second cluster (green) is split, while the right panel is for $K = 5$, when the 4th (yellow) and 5th (cyan) clusters are merged.

Example 15.2 The K-means clustering method is applied to the dataset of handwritten digits from 0 to 9 used previously. The clustering result is visualized in Fig. 15.3, based on the KLT that maps the data samples in the original 256-D space into the 3-D space spanned by the first three eigenvectors with the greatest eigenvalues of the covariance matrix of the dataset. The ground truth labelings are color coded as shown on the left, while the clustering result is shown on the right. We see that the clustering results match the original data reasonably well.

The clustering result can also be shown in the confusion matrix previously used for supervised classification. Here the columns and rows represent the class labelings (the ground truth not used clustering) and the columns represent the clusters based on the clustering result, i.e., the element in the ith row and jth column is the number of samples labeled to belong to the ith class but assigned to the jth cluster. Ideally, there should be only one nonzero value in each row indicating all samples in a class are assigned to the same cluster, and one nonzero value in each column indicating all samples assigned to a cluster are from the same class. Based on this confusion matrix, we can also define the error rate as the ratio between the number of miss-clustered samples, the sum of all components in the confusion matrix excluding the greatest one in each row, and he total number of samples, sum of all components in the matrix.

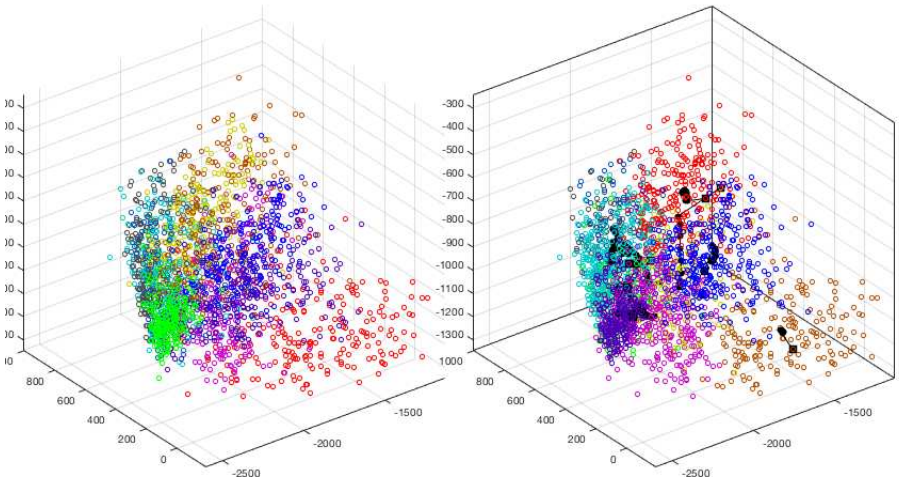


Figure 15.3 K-Means Clustering of Handwritten Digits, ground truth classes (left) vs. clusters (right)

$$\begin{bmatrix} 1 & 0 & 25 & 165 & 27 & 2 & 3 & 0 & 1 & 0 \\ 0 & 209 & 0 & 0 & 2 & 0 & 13 & 0 & 0 & 0 \\ 2 & 3 & 2 & 0 & 8 & 2 & 14 & 151 & 0 & 42 \\ 103 & 0 & 0 & 1 & 5 & 3 & 83 & 0 & 0 & 29 \\ 0 & 28 & 1 & 0 & 15 & 2 & 62 & 0 & 115 & 1 \\ 44 & 0 & 1 & 1 & 167 & 0 & 8 & 0 & 3 & 0 \\ 0 & 6 & 160 & 14 & 44 & 0 & 0 & 0 & 0 & 0 \\ 1 & 6 & 0 & 0 & 3 & 122 & 74 & 2 & 1 & 15 \\ 82 & 1 & 1 & 1 & 33 & 3 & 47 & 2 & 0 & 54 \\ 4 & 2 & 0 & 1 & 0 & 28 & 119 & 1 & 64 & 5 \end{bmatrix} \tag{15.10}$$

We note that as the resulting clusters are in random order unrelated to the ordering of the class identities in the dataset (ground truth not used in clustering), it is in general not the case (as in supervised classification) that the diagonal components of the confusion matrix always take larger values while all off-diagonal components take small values (ideally zero). In clustering, the largest value, assumed to be in the jth column, in each row, assumed to be the ith one, indicates that most of the samples known to belong to the ith class are assigned to the jth cluster. For a good clustering result, there should be only one large value in each row indicating most of the samples in a class are assigned to the same cluster, and there should be only one large value in each column indicating most samples assigned to a cluster are from the same class.

To further visualize the clustering result obtained by the K-means method, we convert all data points from 256-D vectors back into 16×16 image form, and then display the averages of all samples in each of the ten cluster as shown



Figure 15.4 Visualization of the Clusters Found by K-Means

in Fig. 15.4. We see that each of the ten clusters clearly represents one distinct digit.

15.2 Gaussian mixture model

The K-means clustering assigns each data point $\mathbf{x}$ to one of the K clusters based on the Euclidean distances between the point and the means of all clusters, without considering the overall distribution of the data points in each cluster represented by the covariance as well as the mean.

This issue can be addressed by the method of *Gaussian mixture model (GMM)* based on the assumption that each of the K clusters $\{C_1, \cdots, C_K\}$ in the given dataset can be modeled by a Gaussian distribution $\mathcal{N}(\mathbf{m}_k, \mathbf{\Sigma}_k)$, $(k = 1, \cdots, K)$ in terms of the mean vector $\mathbf{m}_k$ and covariance matrix $\mathbf{\Sigma}_k$. The overall distribution of the entire dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ is the weighted sum of the K Gaussian distributions:

$$\begin{aligned} p(\mathbf{x}) &= \sum_{k=1}^{K} P_k \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \mathbf{\Sigma}_k) \\ &= \sum_{k=1}^{K} P_k \left[\frac{1}{(2\pi)^{d/2} |\mathbf{\Sigma}_k|^{1/2}} \exp\left(-\frac{1}{2} (\mathbf{x} - \mathbf{m}_k)^T \mathbf{\Sigma}_k^{-1} (\mathbf{x} - \mathbf{m}_k) \right) \right] \end{aligned} \tag{15.11}$$

where P_k is the weight for the kth Gaussian $\mathcal{N}(\mathbf{m}_k, \mathbf{\Sigma}_k)$, satisfying

$$\int_{-\infty}^{\infty} p(\mathbf{x})\, d\mathbf{x} = \sum_{k=1}^{K} P_k \int_{-\infty}^{\infty} \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \mathbf{\Sigma}_k)\, d\mathbf{x} = \sum_{k=1}^{K} P_k = 1 \tag{15.12}$$

The GMM clustering is a method of *expectation maximization (EM)*, by which all model parameters denoted by $\boldsymbol{\theta} = \{P_k, \mathbf{m}_k, \mathbf{\Sigma}_k,\ (k = 1, \cdots, K)\}$ is first estimated based on the given dataset, and then all data samples in the dataset are assigned to one of the clusters described by these parameters.

We note that the GMM model in Eq. (15.11) is actually the same as Eq. (13.11) in the naive Bayes classification. These two methods are similar in the sense that each cluster or class C_k is modeled by a Gaussian $\mathcal{N}(\mathbf{m}_k, \mathbf{\Sigma}_k)$, weighted by P_k, and the model parameters $\mathbf{m}_k$ and $\mathbf{\Sigma}_k$, as well as P_k, are to be estimated based on the given dataset.

However, the two methods are different in that the dataset $\mathbf{X}$ in the supervised naive Bayes method is labeled by $\mathbf{y}$, which is not available in unsupervised GMM. Instead, here we introduce a latent or hidden variable $\mathbf{z} = [z_1, \cdots, z_K]^T$ as the cluster labeling of each ample $\mathbf{x}$ in the dataset $\mathbf{X}$. All components of $\mathbf{z}_n$ are binary random variables $z_k \in \{0, 1\}$, of which only one is 1, e.g., $z_l = 1$, indicating $\mathbf{x}$ belongs to the lth cluster C_l, while all others are 0. As $\mathbf{x}$ must belong to one of the K clusters, these K binary variables add up to 1, i.e., $\sum_{k=1}^{K} z_k = 1$.

We consider the following probabilities defined for each of the K clusters C_k $(k = 1, \cdots, K)$:

- The *prior probability* P_k, $(k = 1, \cdots, K)$ for any unobserved data sample in the dataset to belong to C_k, they satisfy

$$\sum_{k=1}^{K} P_k = 1 \tag{15.13}$$

  These K prior probabilities add up to 1, i.e, the K events $z_k = 1$ $(k = 1, \cdots, K)$ are mutually exclusive and complementary, as any sample $\mathbf{x}$ belongs to one and only one of the K clusters.

- The probability distribution of all samples in cluster C_k, assumed to be a Gaussian:

$$p(\mathbf{x}|z_k = 1, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \boldsymbol{\Sigma}_k) \tag{15.14}$$

- The joint distribution of $\mathbf{x}$ and $z_k = 1$:

$$p(\mathbf{x}, z_k = 1|\boldsymbol{\theta}) = p(\mathbf{x}|z_k = 1, \boldsymbol{\theta})\ P(z_k = 1) = P_k \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \boldsymbol{\Sigma}_k) \tag{15.15}$$

  where $P(z_{nk} = 1) = P_k$ is the probability for any $\mathbf{x}_n$ to belong to C_k.

Marginalizing this joint probability over the latent variable z_k, we get the Gaussian mixture model, the distribution $p(\mathbf{x})$ of any sample $\mathbf{x}$ regardless to which cluster it belongs:

$$p(\mathbf{x}|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(\mathbf{x}, z_k = 1 fe|\boldsymbol{\theta}) = \sum_{k=1}^{K} p(\mathbf{x}|z_k = 1, \boldsymbol{\theta})\ P(z_k = 1) = \sum_{k=1}^{K} P_k \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \boldsymbol{\Sigma}_k) \tag{15.16}$$

Note that Eqs. (15.13) through (15.16) are the same as Eqs. (13.8) through (13.11) in the naive Bayes classifier, respectively.

All such probabilities defined for $z_k = 1$ can be generalized to $\mathbf{z} = [z_1, \cdots, z_K]^T$

for all K:

$$p(\mathbf{z}|\boldsymbol{\theta}) = \prod_{k=1}^{K} P_k^{z_k} \tag{15.17}$$

$$p(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}, \mathbf{m}_k, \boldsymbol{\Sigma}_k)^{z_k} \tag{15.18}$$

$$p(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta}) = p(\mathbf{z})\ p(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}) = \prod_{k=1}^{K} (P_k \mathcal{N}(\mathbf{m}_k, \boldsymbol{\Sigma}_k))^{z_k} \tag{15.19}$$

We further define $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_N]$ of which the nth column $\mathbf{z}_n = [z_{n1}, \cdots, z_{nK}]^T$ is the labeling of the nth sample $\mathbf{x}_n$ in the dataset $\mathbf{X}$ containing N i.i.d. samples. Note that here $\mathbf{Z}$ is defined in the same way as $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$ in softmax regression (Section 7.2), both treated as the labeling of $\mathbf{X}$, with the only difference that $\mathbf{Y}$ is provided in the training data available for a supervised method, but here $\mathbf{Z}$ is a latent variable, not available in the unlabeled dataset in unsupervised clustering.

For a specific sample $\mathbf{x}_n$ in the dataset, the joint probibility in Eq. (15.19) can be written as

$$p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta}) = \prod_{k=1}^{K} (P_k \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k))^{z_{nk}}, \qquad (n = 1, \cdots, N) \tag{15.20}$$

based on which the likelihood function of the GMM model parameter $\boldsymbol{\theta}$ can be expressed as

$$\begin{aligned} L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Z}) = p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) &= p\left([\mathbf{x}_1, \cdots, \mathbf{x}_N], [\mathbf{z}_1, \cdots, \mathbf{z}_N] \middle| \mathbf{m}_k, \boldsymbol{\Sigma}_k, P_k \ \ (k = 1, \cdots, K)\right) \\ &= \prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta}) = \prod_{n=1}^{N} \prod_{k=1}^{K} (P_k \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k))^{z_{nk}} \end{aligned} \tag{15.21}$$

and the log likelihood is:

$$\begin{aligned} l(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Z}) &= \log L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Z}) = \log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) \\ &= \log \left[\prod_{n=1}^{N} \prod_{k=1}^{K} (P_k \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k))^{z_{nk}} \right] \\ &= \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left[\log P_k + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k) \right] \end{aligned} \tag{15.22}$$

Similar to the general method of maximum likelihood estimation (MLE) that finds the optimal model parameter $\boldsymbol{\theta}$ that maximizes the likelihood $L(\boldsymbol{\theta})$, here the GMM method finds the model parameter $\boldsymbol{\theta} = \{P_k, \mathbf{m}_k, \boldsymbol{\Sigma}_k,\ (k = 1, \cdots, K)\}$ that maximizes the expectation of the likelihood function $l(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Z})$. However, as GMM is an unsupervised method, the ground truth labeling is not available in the given dataset, it has to be carried out in two separate steps. First, in

the E-step the expectation of the log likelihood function is estimated based on the previously estimated model parameters; second, in the M-step the estimated parameters that maximize the expectation of the log likelihood are obtained and the previous estimation is updated. As these two steps are interdependent of each other, they need to be carried out alternatively in an iteration from some initial guess of the parameters. This process is called an EM method due to these E and M steps.

Now we consider the specific computations taking place in each of the two steps.

- **E-step:** Find the expectation of the log likelihood function.

  We first find the posterior probability for an observed $\mathbf{x}_n$ to belong to cluster C_k, indicated by $z_{nk} = 1$ (and $z_{nl} = 0$ for all $l \neq k$):

$$P_{nk} = P(z_{nk} = 1|\mathbf{x}_n, \boldsymbol{\theta}) = \frac{p(\mathbf{x}_n, z_{nk} = 1|\boldsymbol{\theta})}{p(\mathbf{x}_n|\boldsymbol{\theta})} = \frac{P_k \mathcal{N}(\mathbf{x}_n; \mathbf{m}_k \boldsymbol{\Sigma}_k)}{\sum_{l=1}^{K} P_l \mathcal{N}(\mathbf{x}_n; \mathbf{m}_l \boldsymbol{\Sigma}_l)}$$
$$(n = 1, \cdots, N;\ k = 1, \cdots, K) \tag{15.23}$$

  As $\mathbf{x}_n$ has to belong to one of the K clusters, these K posterior probabilities add up to 1, same as the prior probabilities $P(z_k = 1) = P_k$:

$$\sum_{k=1}^{K} P(z_k = 1) = \sum_{k=1}^{K} P_k = 1, \qquad \sum_{k=1}^{K} P(z_{nk} = 1|\mathbf{x}_n, \boldsymbol{\theta}) = \sum_{k=1}^{K} P_{nk} = 1 \tag{15.24}$$

  Once this posterior $P_{nk} = p(z_{nk} = 1|\mathbf{x}_n, \boldsymbol{\theta})$ is calculated based on Eq. (15.23) for all $n = 1, \cdots, N$), they are used to assign $\mathbf{x}_n$ to cluster C_k if $P_{nk} = \max_l P_{nl}$.

  We note that the definition of P_{nk} is similar to the softmax function s_{nk} with the only difference that here the weighted Gaussian function are used instead of the softmax function used in softmax regression.

  We also note that the posterior probability P_{nk} defined above represents a *soft* decision, in the sense that it is possible for any $\mathbf{x}_n$ to belong to any C_k with probability $0 < P_{nk} < 1$ for all $k = 1, \cdots, K$, instead of a *hard* decision, in the sense that $\mathbf{x}_n$ belongs to only one specific C_k with $P_{nk} = 1$, while $P_{nl} = 0$ for all $l \neq k$, as in K-means clustering.

  We can now find the expectation of the log likelihood with respect to the latent variables in $\mathbf{Z}$:

$$\begin{aligned} E_Z\left(\log L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Z})\right) &= E_Z\left[\sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk}\left[\log P_k + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k)\right]\right] \\ &= \sum_{n=1}^{N}\sum_{k=1}^{K} E_Z(z_{nk})\left[\log P_k + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k)\right] \\ &= \sum_{n=1}^{N}\sum_{k=1}^{K} P_{nk}\left[\log P_k + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k)\right] \end{aligned} \tag{15.25}$$

The last equality is due to the following:

$$E_z(z_{nk}) = 1\ P(z_{nk}=1|\mathbf{x}_n) + 0\ P(z_{nk}=0|\mathbf{x}_n) = P(z_{nk}=1|\mathbf{x}_n) = P_{nk} \tag{15.26}$$

- **M-step:** Find the optimal model parameters that maximize the expectation of the log likelihood in Eq. (15.25).

  We first set to zero the derivatives of the expectation of the log likelihood with respect to each of the parameters in $\boldsymbol{\theta} = \{P_k,\ \mathbf{m}_k,\ \boldsymbol{\Sigma}_k,\ (k = 1, \cdots, K)\}$, and then solve the resulting equations to get the optimal parameters.

  – Find P_k:

    As all K of P_k's need to satisfy the constraint $\sum_{k=1}^{K} P_k = 1$, we first construct the Lagrangian function composed of an extra Lagrange multiplier term for the constraint as well as the log likelihood as the objective function:

$$L(\boldsymbol{\theta},\ \lambda) = \sum_{n=1}^{N}\sum_{k=1}^{K} P_{nk}\left[\log P_k + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k)\right] + \lambda\left(\sum_{k=1}^{K} P_k - 1\right) \tag{15.27}$$

    and set it derivative with respect to P_k to zero:

$$\begin{aligned}\frac{\partial}{\partial P_k} L(\boldsymbol{\theta},\ \lambda) &= \frac{\partial}{\partial P_k}\left[\sum_{n=1}^{N}\sum_{k=1}^{K} P_{nk}\left[\log P_k + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k)\right] + \lambda\left(\sum_{k=1}^{K} P_k - 1\right)\right] \\ &= \sum_{n=1}^{N} P_{nk}\frac{1}{P_k} + \lambda = 0\end{aligned} \tag{15.28}$$

    Multiplying both sides by P_k, we get

$$\sum_{n=1}^{N} P_{nk} + P_k\lambda = N_k + P_k\lambda = 0 \tag{15.29}$$

    where we have defined

$$N_k = \sum_{n=1}^{N} P_{nk} \tag{15.30}$$

    that satisfies

$$\sum_{k=1}^{K} N_k = \sum_{k=1}^{K}\left(\sum_{n=1}^{N} P_{nk}\right) = \sum_{n=1}^{N}\left(\sum_{k=1}^{K} P_{nk}\right) = \sum_{n=1}^{N} 1 = N \tag{15.31}$$

    Summing Eq. (15.29) over all $k = 1, \cdots, K$, we get

$$\sum_{k=1}^{K}(N_k + P_k\lambda) = \sum_{k=1}^{K} N_k + \lambda\left(\sum_{k=1}^{K} P_k\right) = N + \lambda = 0 \tag{15.32}$$

Substituting $\lambda = -N$ back into Eq. (15.29), we get the expression for the prior

$$p(z_k = 1) = P_k = \frac{N_k}{N} = \frac{1}{N}\sum_{n=1}^{N} P_{nk} \tag{15.33}$$

This is the same as the prior P_k in Eq. (13.8) in naive Bayes classification, but here N_k defined in Eq. (15.30) is the sum of the probabilities for all N data samples to belong to C_k, instead of the number of data samples in C_k (unknown in this unsupervised case).

– Find $\mathbf{m}_k$:

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{m}_k} E_Z(\log L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Z})) &= \frac{\partial}{\partial \mathbf{m}_k}\left[\sum_{n=1}^{N}\sum_{l=1}^{K} P_{nl}\left[\log P_l + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_l, \boldsymbol{\Sigma}_l)\right]\right] \\
&= \sum_{n=1}^{N} P_{nk}\frac{\partial}{\partial \mathbf{m}_k}\log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k) \\
&\stackrel{*}{=} \sum_{n=1}^{N} P_{nk}\frac{\partial}{\partial \mathbf{m}_k}\left[-\frac{1}{2}(\mathbf{x}_n - \mathbf{m}_k)^T\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \mathbf{m}_k)\right] \\
&= \frac{1}{2}\sum_{n=1}^{N} P_{nk}\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \mathbf{m}_k) = \mathbf{0}
\end{aligned} \tag{15.34}$$

Note that here $\stackrel{*}{=}$ indicates that we have neglected the scaling coefficient $(2\pi)^{-d/2}\,|\boldsymbol{\Sigma}_k|^{-1/2}$ of the Gaussian, which is independent of $\mathbf{m}_k$. Multiplying $2\boldsymbol{\Sigma}_k$ on both sides, we get

$$\sum_{n=1}^{N} P_{nk}(\mathbf{x}_n - \mathbf{m}_k) = \sum_{n=1}^{N} P_{nk}\mathbf{x}_n - \sum_{n=1}^{N} P_{nk}\mathbf{m}_k = \sum_{n=1}^{N} P_{nk}\mathbf{x}_n - N_k\mathbf{m}_k = \mathbf{0} \tag{15.35}$$

Solving for $\mathbf{m}_k$ we get

$$\mathbf{m}_k = \frac{1}{N_k}\sum_{n=1}^{N} P_{nk}\mathbf{x}_n \tag{15.36}$$

– Find $\boldsymbol{\Sigma}_k$:

$$\frac{\partial}{\partial \boldsymbol{\Sigma}_k} E_Z(\log L(\boldsymbol{\theta}|\mathbf{X},\mathbf{Z})) = \frac{\partial}{\partial \boldsymbol{\Sigma}_k}\left[\sum_{n=1}^{N}\sum_{l=1}^{K} P_{nl}\left[\log P_l + \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_l, \boldsymbol{\Sigma}_l)\right]\right]$$

$$= \sum_{n=1}^{N} P_{nk} \frac{\partial}{\partial \boldsymbol{\Sigma}_k} \log \mathcal{N}(\mathbf{x}_n, \mathbf{m}_k, \boldsymbol{\Sigma}_k)$$

$$= -\frac{1}{2}\sum_{n=1}^{N} P_{nk}\left[\frac{\partial}{\partial \boldsymbol{\Sigma}_k}\log|\boldsymbol{\Sigma}_k| + \frac{\partial}{\partial \boldsymbol{\Sigma}_k}(\mathbf{x}_n - \mathbf{m}_k)^T \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \mathbf{m}_k)\right]$$

$$= -\frac{1}{2}\sum_{n=1}^{N} P_{nk}\left[\boldsymbol{\Sigma}_k^{-1} - \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^T \boldsymbol{\Sigma}_k^{-1}\right] = \mathbf{0} \tag{15.37}$$

Here we have used the following facts (for more details see *Matrix Cookbook* `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`)

$$\frac{d}{d\mathbf{A}}\log|\mathbf{A}| = (\mathbf{A}^{-1})^T, \qquad \frac{d}{d\mathbf{A}}\left(\mathbf{a}^T\mathbf{A}^{-1}\mathbf{b}\right) = -(\mathbf{A}^{-1})^T\mathbf{a}\mathbf{b}^T(\mathbf{A}^{-1})^T \tag{15.38}$$

Pre and post multiplying $\boldsymbol{\Sigma}_k$ on both sides of the equation above we get

$$\sum_{n=1}^{N} P_{nk}\left(\boldsymbol{\Sigma}_k - (\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^T\right) = \mathbf{0} \tag{15.39}$$

Solving for $\boldsymbol{\Sigma}_k$, we get

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k}\sum_{n=1}^{N} P_{nk}(\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^T \tag{15.40}$$

We note that all model parameters in $\boldsymbol{\theta} = \{P_k, \mathbf{m}_k, \boldsymbol{\Sigma}_k, (k = 1, \cdots, K)\}$ found respectively in Eqs. (15.33), (15.36), and (15.40) in the M-step depend on P_{nk} in Eq. (15.23) in the E-step, which in turn is a function of these parameters, i.e., the E-step and M-step need to be carried out in an alternative and iterative fashion from some initial values of the parameters until convergence.

In summary, here are the steps of GMM clustering algorithm:

1. Initialize means $\mathbf{m}_k$, covariance $\boldsymbol{\Sigma}_k$ and coefficient P_k.
2. The E-step:

   Find P_{nk} for all N data points and all K clusters and then N_k:

$$P_{nk} = P(r_k = 1|\mathbf{x}_n) = \frac{P_k\,\mathcal{N}(\mathbf{x}_n; \mathbf{m}_k, \boldsymbol{\Sigma}_k)}{\sum_{l=1}^{K} P_l\,\mathcal{N}(\mathbf{x}_n; \mathbf{m}_l, \boldsymbol{\Sigma}_l)}, \qquad N_k = \sum_{n=1}^{N} P_{nk} \tag{15.41}$$

3. The M-step:

Recalculate the parameters that maximize the likelihood function:

$$P_k = \frac{N_k}{N}$$
$$\mathbf{m}_k = \frac{1}{N_k}\sum_{n=1}^{N} P_{nk}\mathbf{x}_n$$
$$\mathbf{\Sigma}_k = \frac{1}{N_k}\sum_{n=1}^{N} P_{nk}(\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^T \tag{15.42}$$

4. Terminate the iteration if the parameters have converged, i.e., the difference between their values of two consecutive iterations is sufficiently small. Otherwise go back to step 2. The probability for each sample $\mathbf{x}_n$ to belong to cluster C_l is $p_{nl} = P(z_{nl} = 1|\mathbf{x}_n, \boldsymbol{\theta})$, and it is therefore assigned to C_k if $P_{nk} = \max_l p_{nl}$.

The Matlab code below is the essential part of the algorithm including the initialization of parameters and the iteration loop. The function `mvnpdf(X,mk(:,k),Ck(:,:,k))` evaluate the Gaussian pdf of the kth cluster $\mathcal{N}(\mathbf{x}_n; \mathbf{m}_k, \mathbf{\Sigma}_k)$ at all N samples in dataset `X`.

```
[N d]=size(X);                % N samples of d dimensions
mk=zeros(d,K);                % old mean vectors
Mk=zeros(d,K);                % new mean vectors
Ck=zeros(d,d,K);              % covariance matrices
p=zeros(N,K);                 % posterior
Pnk=zeros(N,K);               % Pnk
Pk=zeros(K,1);                % prior Pk=Nk/N
Nk=zeros(1,K);                % number of samples in kth cluster
for k=1:K                     % initial parameters for kth cluster
    sd=randi(N,1,2*d);        % random indices of subsamples
    mk(:,k)=mean(X(sd,:))';   % mean of these samples
    Ck(:,:,k)=cov(X(sd,:));   % covariance of these samples
    Pk(k)=1/K;                % prior probability
end
er=1;
while er>10^(-4)
    for k=1:K                 % for each of K clusters
        p(:,k)=Pk(k)*mvnpdf(X,mk(:,k),Ck(:,:,k));  % weighted posterior
    end
    D=sum(p,2);               % sum over all K terms
    er=0;
    for k=1:K                 % update parameters for kth cluster
        Pnk(:,k)=p(:,k)./D;   %
        Nk(k)=sum(Pnk(:,k)); % Nk
        Pk(k)=Nk(k)/N;        % Pk
```

```
        if Nk(k)>0
            Mk(:,k)=sum(X.*repmat(Pnk(:,k),[1,d]),1)/Nk(k);   % mean
            Xm=X-repmat(Mk(:,k)',[N,1]);
            Ck(:,:,k)=Xm'*(Xm.*repmat(Pnk(:,k),[1,d]))/Nk(k); % covariance
        end
        er=er+norm(mk(:,k)-Mk(:,k));  % difference between two iterations
    end
    mk=Mk;                          % keep iterating
end
```

We can show that the K-means algorithm is actually a special case of the EM algorithm, when all covariance matrices are the same $\boldsymbol{\Sigma}_k = \sigma^2\mathbf{I}$, and σ^2 approaches to zero. In this case we have:

$$p(\mathbf{x}|z_k = 1, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}|\mathbf{m}_k, \sigma^2\mathbf{I}) = \frac{1}{(2\pi)^{d/2}\sigma} \exp\left(-\frac{1}{2\sigma^2}||\mathbf{x} - \mathbf{m}_k||^2\right) \quad (15.43)$$

and the probability for any $\mathbf{x}_n \in \mathbf{X}$ to belong to cluster C_k is:

$$P_{nk} = P(z_k = 1|\mathbf{x}_n, \boldsymbol{\theta}) = \frac{P_k\mathcal{N}(\mathbf{x}_n; \mathbf{m}_l\boldsymbol{\Sigma}_l)}{\sum_{l=1}^K P_l\mathcal{N}(\mathbf{x}_n; \mathbf{m}_l\boldsymbol{\Sigma}_l)} = \frac{P_k \exp(-||\mathbf{x}_n - \mathbf{m}_k||^2/2\sigma^2)}{\sum_{l=1}^K P_l \exp(-||\mathbf{x}_n - \mathbf{m}_l||^2/2\sigma^2)} \quad (15.44)$$

When $\sigma^2 \to 0$, all terms in the denominator approach to zero, but the one with minimum $||\mathbf{x} - \mathbf{m}_l||$ approaches to zero most slowly, and becomes the dominant term of the denominator. If the numerator happens to be this term, i.e., $k = l$, then $P_{nk} = 1$; otherwise the numerator approaches zero and $P_{nk} = 0$. Now P_{nk} defined above becomes:

$$\begin{aligned} \lim_{\sigma^2\to 0} P_{nk} &= \lim_{\sigma^2\to 0} \frac{P_k \exp(-||\mathbf{x}_n - \mathbf{m}_k||^2/2\sigma^2)}{\sum_{l=1}^K P_l \exp(-||\mathbf{x}_n - \mathbf{m}_l||^2/2\sigma^2)} \\ &= \begin{cases} 1 & \text{if } ||\mathbf{x}_n - \mathbf{m}_k|| = \min_l ||\mathbf{x}_n - \mathbf{m}_l|| \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (15.45)$$

Now the posterior probability $0 < P_{nk} < 1$ defined in Eq. (15.23) for a soft decision becomes a binary value $P_{nk} \in \{0,\ 1\}$ for a hard binary decision to assign $\mathbf{x}_n$ to C_k with the smallest distance. Also $N_k = \sum_{n=1}^N P_{nk}$ defined in Eq. (15.30) as the sum of the posterior probabilities for all N data points to belong to C_k becomes N_k as the number of data samples assigned only to C_k. In other words, now the probabilistic EM method based on both $\mathbf{m}$ and $\boldsymbol{\Sigma}$ becomes the deterministic K-means method based on $\mathbf{m}$ only.

We can also make a comparison between the GMM method for unsupervised clustering and the softmax regression for supervised classification. First, the latent variables $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_N]$ in GMM play a similar role as the labeling $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$ in softmax regression multiclass classification. However, the difference is that $\mathbf{Y}$ is explicitly given in the training set for a supervised classification, while $\mathbf{Z}$ is hidden for an unsupervised clustering analysis. Second, we note that the probability $P_{nk} = p(z_{nk} = 1|\mathbf{x}_n, \boldsymbol{\theta})$ given in Eq. (15.23) is similar

to the softmax function $\phi_{nk} = P(y' = k|\mathbf{x}_n)$ in the softmax method in terms of their form, with the only difference that the Gaussian function is used for GMM while the exponential function is used for softmax.

Example 15.3 Both the K-means and GMM clustering methods are tested based on the same synthetic dataset generated by the following Matlab code

```
mk=zeros(d,3);
Ck=zeros(d,d,3);
Ck(:,:,1)=[1 1.5; 1.5 5];   mk(:,1)=[5; 4];
Ck(:,:,2)=[3 -1.5; -1.5 1]; mk(:,2)=[6; 5];
Ck(:,:,3)=[2 1; 1 2];       mk(:,3)=[10; 2];
Ki=[200 200 200];
X1=mvnrnd(mk(:,1)',Ck(:,:,1),Ki(1));
X2=mvnrnd(mk(:,2)',Ck(:,:,2),Ki(2));
X3=mvnrnd(mk(:,3)',Ck(:,:,3),Ki(3));
```

This code generates three clusters of normally distributed data points. Note that the means of the first two clusters are very close to each other, but their covariances are such that the their distributions overlap with each other but are in very different orientations.

The intermediate results after each iteration are shown in the panels in Fig. 15.5 for the K-means method and Fig. 15.6 for the EM clustering method. The data points are color coded according to their class identity (ground truth not used in clustering), and the contour lines for each of the clusters are also shown based on the mean and covariance of all sampled assigned to each cluster in the current iteration. Comparing the two sets of results obtained by the two methods, we see that the K-means method cannot separate the first two clusters (in red and green), while the third cluster (in blue) is separated incorrectly into two clusters. On the other hand, the GMM method can correctly separate all three clusters.

Example 15.4 The two clustering methods are also applied to the 4-D Iris dataset containing three classes each of 50 sample vectors. The PCA method is used to visualize the first two principal components, as shown in Figs. 15.7 and 15.8. Also, as can be seen from their confusion matrices in Eq. (15.46), the error rate of the EM method is 5/150, much smaller than 18/15 of the K-means method.

$K-means$		
0	0	50
50	0	0
18	32	0

EM		
0	0	50
45	5	0
0	50	0

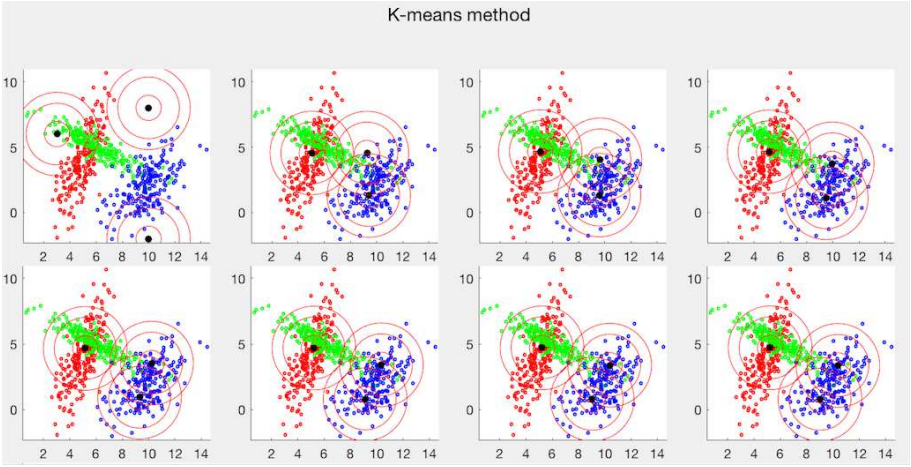


Figure 15.5 Clustering by K-Means Method (first 8 iterations)

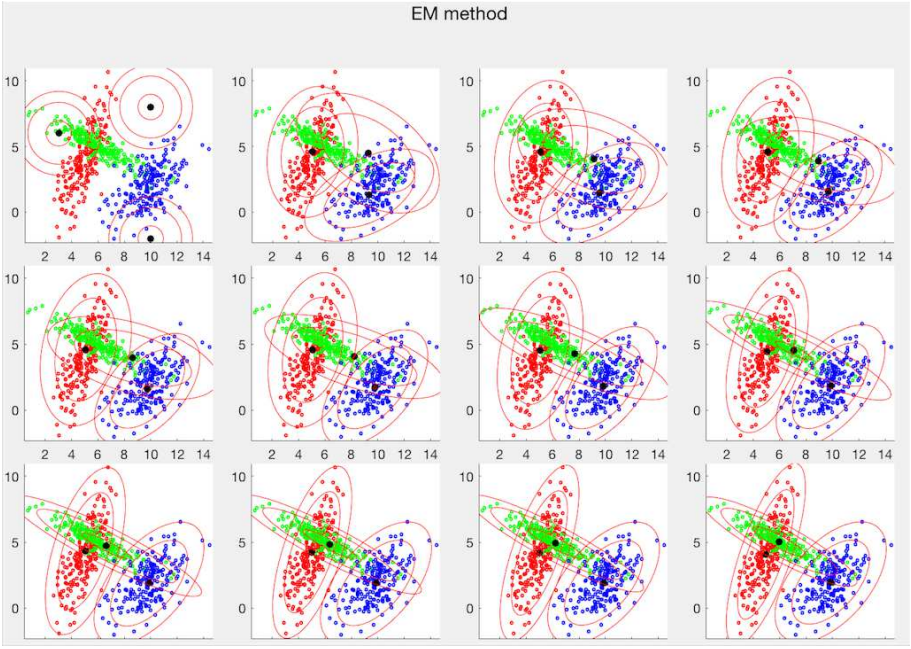


Figure 15.6 Clustering by Gaussian Mixture Model (first 12 iterations)

15.3 Bernoulli Mixture Model

If the data points are binary, i.e., each data point is treated as a binary random variable that takes either of two values such as 1 and 0 with probabilities μ and $1-\mu$, then the assumed Gaussian distribution is no longer valid and the Gaussian

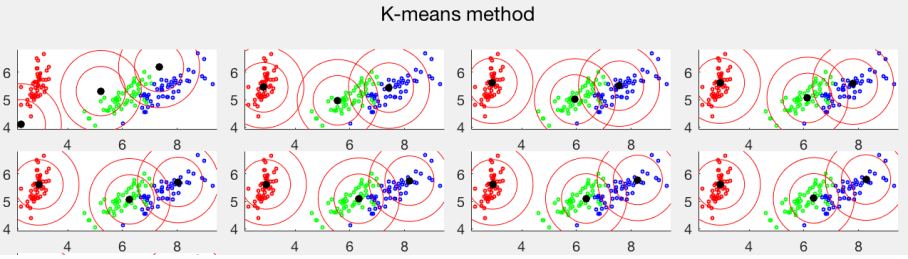


Figure 15.7 Clustering of Iris Dataset by K-Means Method

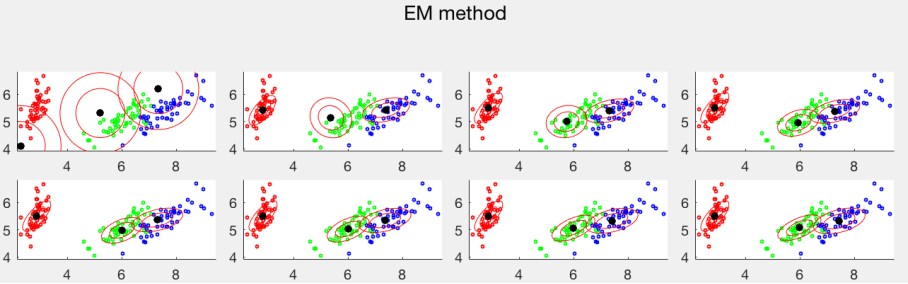


Figure 15.8 Clustering of Iris Dataset by Gaussian Mixture Model

mixture model is not suitable. In this case, we can modify the EM method for clustering based on *Bernoulli mixture model (BMM)* instead of GMM.

The *probability mass function (pmf)* of the Bernoulli distribution can be used instead:

$$\mathcal{B}(x,\mu) = \mu^x(1-\mu)^{1-x} = \begin{cases} \mu & \text{if } x=1 \\ 1-\mu & \text{if } x=0 \end{cases} \tag{15.46}$$

The mean and variance of x are

$$E[x] = 1\ P(x=1) + 0\ P(x=0) = 1\ \mu + 0\ (1-\mu) = \mu \tag{15.47}$$

$$\begin{aligned} Var[x] &= E[(x-E[x])^2] = E[x^2] - E[x]^2 \\ &= 1^2\ P(x=1) + 0^2\ P(x=0) - \mu^2 = \mu - \mu^2 = \mu(1-\mu) \end{aligned} \tag{15.48}$$

Note that as the variance is solely determined by the mean $E[x] = \mu$ it does not need to be estimated separately, therefore the method based on BMM is simpler than that based on GMM.

A set of d independent binary variables can be represented as a random vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ with mean vector and covariance matrix as shown below:

$$E[\mathbf{x}] = \mathbf{m} = [\mu_1, \cdots, \mu_N]^T \tag{15.49}$$

$$\text{Cov}[\mathbf{x}] = \mathbf{\Sigma} = \text{diag}(\mu_i(1-\mu_i)) = \begin{bmatrix} \mu_1(1-\mu_1) & & 0 \\ & \ddots & \\ 0 & & \mu_d(1-\mu_d) \end{bmatrix} \tag{15.50}$$

The off-diagonal covariances are zero as the d variables are assumed to be independent of each other. The joint pmf of the binary random vector $\mathbf{x}$ is

$$\mathcal{B}(\mathbf{x}, \mathbf{m}) = \prod_{i=1}^{d} \mathcal{B}(x_i|\mu_i) = \prod_{i=1}^{d} \mu_i^{x_i}(1-\mu_i)^{1-x_i} \tag{15.51}$$

and the log pmf:

$$\log \mathcal{B}(\mathbf{x}, \mathbf{m}) = \log\left(\prod_{i=1}^{d} \mathcal{B}(x_i, \mu_i)\right) = \sum_{i=1}^{d} [x_i \log \mu_i + (1-x_i)\log(1-\mu_i)] \tag{15.52}$$

Similar to the Gaussian mixture model, the Bernoulli mixture model of K multivariate Bernoulli distributions is defined as:

$$\begin{aligned} p(\mathbf{x}|\boldsymbol{\theta}) &= p(\mathbf{x}|\mathbf{m}_k, P_k, (k=1,\cdots,K)) = \sum_{k=1}^{K} P_k \mathcal{B}(\mathbf{x}, \mathbf{m}_k) \\ &= \sum_{k=1}^{K} P_k \prod_{i=1}^{d} \mu_{ki}^{x_i}(1-\mu_{ki})^{1-x_i} \end{aligned} \tag{15.53}$$

where $\boldsymbol{\theta} = \{\mathbf{m}_k, P_k, (k=1,\cdots,K)\}$ denotes all parameters of the mixture model to be estimated based on the given dataset. The mean of this mixture model is

$$\mathbf{m} = E[\mathbf{x}] = \sum_{k=1}^{K} P_k E_k[\mathbf{x}] = \sum_{k=1}^{K} P_k \mathbf{m}_k \tag{15.54}$$

Also similar to the Gaussian mixture model, we introduce a set of K latent binary random variables $\mathbf{z} = [z_1, \cdots, z_K]^T$ with binary components $z_k \in \{0,\ 1\}$ satisfying $\sum_{k=1}^{K} z_k = 1$, and get the prior probability of $\mathbf{z}$, the conditional probability of $\mathbf{x}$ given $\mathbf{z}$, and the joint probability of $\mathbf{x}$ and $\mathbf{z}$ as the following

$$p(\mathbf{z}|\boldsymbol{\theta}) = \prod_{k=1}^{K} P_k^{z_k} \tag{15.55}$$

$$p(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}) = \prod_{k=1}^{K} \mathcal{B}(\mathbf{x}, \mathbf{m}_k)^{z_k} \tag{15.56}$$

$$p(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta}) = p(\mathbf{z}|\boldsymbol{\theta})\ p(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}) = \prod_{k=1}^{K} \left(P_k\ \mathcal{B}(\mathbf{x}, \mathbf{m}_k)\right)^{z_k} \tag{15.57}$$

Given the dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ containing N i.i.d. samples, we introduce the corresponding latent variables in $\mathbf{Z} = [\mathbf{z}_1, \cdots, \mathbf{z}_N]$, of which each $\mathbf{z}_n = [z_{n1}, \cdots, z_{nK}]^T$ is for the labeling of $\mathbf{x}_n$. Then we can find the likelihood function of the Bernoulli mixture model parameters $\boldsymbol{\theta} = \{P_k, \mathbf{m}_k,\ (k=1,\cdots,K)\}$:

$$\begin{aligned} L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Z}) = p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) &= p([\mathbf{x}_1, \cdots, \mathbf{x}_N], [\mathbf{z}_1, \cdots, \mathbf{z}_N]|\mathbf{m}_k, P_k(k=1,\cdots,K)) \\ &= \prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta}) = \prod_{n=1}^{N}\prod_{k=1}^{K} \left(P_k \mathcal{B}(\mathbf{x}_n, \mathbf{m}_k)\right)^{z_{nk}} \end{aligned} \tag{15.58}$$

and the log likelihood function:

$$\log\ L(\boldsymbol{\theta}|\mathbf{X},\mathbf{Z}) = \log p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}) = \log \prod_{n=1}^{N}\prod_{k=1}^{K} \left(P_k \mathcal{B}(\mathbf{x}_n,\mathbf{m}_k)\right)^{z_{nk}}$$
$$= \sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk}\left[\log P_k + \log \mathcal{B}(\mathbf{x}_n,\mathbf{m}_k)\right] \tag{15.59}$$

Based on the same EM method used in Gaussian mixture model, we can find the optimal parameters that maximize the expectation of the log likelihood function in the following two steps:

- **E-step:** Find the expectation of the likelihood function.

  We first find the posterior probability for any sample $\mathbf{x}_n$ to belong to cluster C_k, denoted by P_{nk}:

$$P_{nk} = P(z_{nk}=1|\mathbf{x}_n,\boldsymbol{\theta}) = \frac{p(\mathbf{x}_n, z_{nk}=1|\boldsymbol{\theta})}{p(\mathbf{x}_n|\boldsymbol{\theta})} = \frac{P_k\, \mathcal{B}(\mathbf{x}_n;\mathbf{m}_k)}{\sum_{l=1}^{K} P_l\, \mathcal{B}(\mathbf{x}_n;\mathbf{m}_l)}$$
$$(n=1,\cdots,N;\ \ k=1,\cdots,K) \tag{15.60}$$

  which is the expectation of z_{nk} (same as Eq. (15.26)):

$$E[z_{nk}] = 1\ P(z_{nk}=1|\mathbf{x}_n) + 0\ P(z_{nk}=0|\mathbf{x}_n) = P(z_{nk}=1|\mathbf{x}_n) = P_{nk} \tag{15.61}$$

  Now we can find the expectation of the log likelihood with respect to the latent variables in $\mathbf{Z}$:

$$E_z\left(\log L(\boldsymbol{\theta}|\mathbf{X},\mathbf{Z})\right) = E_z \sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk}\left[\log P_k + \log \mathcal{B}(\mathbf{x}_n,\mathbf{m}_k)\right]$$
$$= \sum_{n=1}^{N}\sum_{k=1}^{K} E[z_{nk}]\left[\log P_k + \log \prod_{i=1}^{d} \mu_{ki}^{x_{ni}}(1-\mu_{ki})^{1-x_{ni}}\right]$$
$$= \sum_{n=1}^{N}\sum_{k=1}^{K} P_{nk}\left[\log P_k + \sum_{i=1}^{d}\left[x_{ni}\log\mu_{ki} + (1-x_{ni})\log(1-\mu_{ki})\right]\right] \tag{15.62}$$

- **M-step:** Find the optimal model parameters that maximize the expectation of the log likelihood function.

  We first set to zero the derivatives of the expectation of the log likelihood with respect to each of the parameters in $\boldsymbol{\theta} = \{P_k,\ \mathbf{m}_k\ (k=1,\cdots,K)\}$, and then solve the resulting equations to get the optimal parameters.

  – Find P_k: same as in the case of the GMM model:

$$P_k = \frac{N_k}{N} = \frac{1}{N}\sum_{n=1}^{N} P_{nk} \tag{15.63}$$

– Find $\mathbf{m}_k$:

$$\begin{aligned}
&\frac{\partial}{\partial \mathbf{m}_k} E_{\mathbf{Z}} \left(\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})\right) \\
&= \frac{\partial}{\partial \mathbf{m}_k} \sum_{n=1}^{N} \sum_{k=1}^{K} P_{nk} \left[\log P_k + \sum_{i=1}^{d} [x_{ni} \log \mu_{ki} + (1 - x_{ni}) \log(1 - \mu_{ki})]\right] \\
&= \sum_{n=1}^{N} P_{nk} \frac{\partial}{\partial \mathbf{m}_k} \sum_{i=1}^{d} [x_{ni} \log \mu_{ki} + (1 - x_{ni}) \log(1 - \mu_{ki})] = \mathbf{0} \qquad (15.64)
\end{aligned}$$

The ith component of the equation is

$$\begin{aligned}
&\sum_{n=1}^{N} P_{nk} \frac{d}{d\mu_{ki}} [x_{ni} \log \mu_{ki} + (1 - x_{ni}) \log(1 - \mu_{ki})] \\
&= \sum_{n=1}^{N} P_{nk} \left(\frac{x_{ni}}{\mu_{ki}} - \frac{1 - x_{ni}}{1 - \mu_{ki}}\right) = 0 \qquad (15.65)
\end{aligned}$$

i.e.,

$$(1 - \mu_{ki}) \sum_{n=1}^{N} P_{nk} x_{ni} = \mu_{ki} \sum_{n=1}^{N} P_{nk}(1 - x_{ni}) = \mu_{ki} N_k - \mu_{ki} \sum_{n=1}^{N} P_{nk} x_{ni} \qquad (15.66)$$

Solving for μ_{ki} we get

$$\mu_{ki} = \frac{1}{N_k} \sum_{n=1}^{N} P_{nk} x_{ni} \qquad (i = 1, \cdots, d) \qquad (15.67)$$

or, in vector form,

$$\mathbf{m}_k = \frac{1}{N_k} \sum_{n=1}^{N} P_{nk} \mathbf{x}_n \qquad (15.68)$$

The Matlab code shown below is the iteration part of the algorithm. Same as in the code the Gaussian mixture model, all parameters are initialized in a similar manner.

```
er=1;
while er<0.01
   for k=1:K
       p(:,k)=pi(k)*Bpdf(X,mk(:,k)');
   end
   er=0;
   D=sum(p,2);
   for k=1:K
       Pnk(:,k)=p(:,k)./D;
       Nk(k)=sum(Pnk(:,k));
       pi(k)=Nk(k)/N;
```

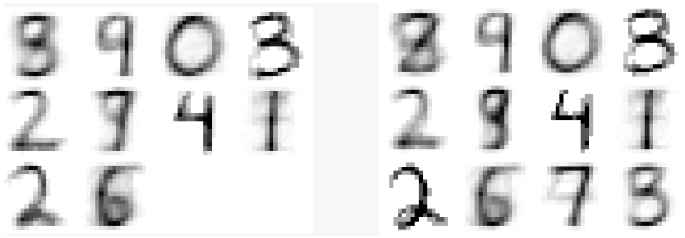


Figure 15.9 Clustering of Handwritten Digits (left $K = 10$, right $K = 12$)

```
        if Nk(k)>0
            Mk(:,k)=sum(X.*repmat(Pnk(:,k),[1,d]),1)/Nk(k);
            Xm=X-repmat(Mk(:,k)',[N,1]);
        end
        er=er+norm(mk(:,k)-Mk(:,k));
    end
end
```

Example 15.5 The clustering algorithm based on the mixture of Bernoulli model is applied to the dataset of handwritten digits after the gray scale images (in the range from 0 to 255) are converted to binary data by a threshold value (e.g., 128). Once the iteration is converged, the final clustering result is shown in the form of the means vectors of the K clusters (no longer binary) in Fig. 15.9 with $K = 10$ (left) and $K = 12$ (right).

Problems

1. Implement K-means clustering algorithm and apply it to the iris data. Try different values for K, such as $K = 2$ and $K = 4$ in addition to the ground truth value $K = 3$.

   Show your results as a K by K confusion matrix and the error rate. Also visualize your result in both 2-D and 3-D spaces after the 4-D dataset is KLT transformed into a lower dimensional space. Color code all data points according to the cluster they are assigned to.
2. Repeat the above for the handwritten digit dataset. Again try different values of K, such as 9 and 11, in addition to the ground truth value $K = 10$.

   Further more, convert the K mean vectors of all samples assigned to the same clusters into 16×16 images, and display these images to see what the clusters actually represents, and if they each resemble one of the digits, same as those shown in Example 15.2.
3. Implement GMM clustering and apply it with the K-means method to the dataset used in Example 15.3 (generated by the code provided in the example). Visualize the results by both GMM and K-means methods with all data points

color coded, and compare the confusion matrices of both methods to confirm GMM achieves much better results than K-mean in this particular case.

4. Apply the GMM method to both the iris and handwritten datasets. For each dataset, show your clustering result in confusion matrix and visualize the color coded data points in 2-D and 3-D spaces. Compare your results with those obtained by the K-means method in the first two problems.
5. Implement clustering based on mixture of Bernoulli model, then apply it to the binary version of the handwritten digit dataset obtained by thresholding all samples and converting them into binary numbers 0 or 1. Compare your results with $K = 10$ with those shown in Fig. 15.9

16 Hierarchical Classifiers

Both supervised classification and unsupervised clustering can be carried out hierarchically in the form of a dendrogram, a tree-like structure, very much like the classifications of different taxonomic ranks in biology (domain, kingdom, phylum, class, order, family, genus, and species). Such a hierarchical classification or clustering method is essentially a divide-and-conquer process by which a complex problem is converted into a sequence of simpler subproblems that can be solved more effectively and efficiently. Certain advantages can be gained by carrying out classification or clustering in a hierarchical manner, in comparison to most of the algorithms previously discussed all attempting to achieve the goal in a single step.

Before the considering various hierarchical algorithms specifically, we note that as a general method for classification and clustering, a decision tree may grow very deep to accommodate certain detailed variations in the data, and consequently overfitting the data. This problem can be addressed by the method of *random forest* which takes the average of multiple decision trees each trained on different subsets of the training dataset. Random forest is similar to the K-fold cross-validation in the sense that the given dataset is subdivided into multiple parts for more reliable outcomes.

16.1 Bottom-Up vs Top-Down Methods

The tree structure for a hierarchical method is typically constructed in either bottom-up fashion by merging subgroups of data points into bigger groups, or top-down fashion by splitting big groups into smaller subgroups. In the following, we will concentrate on how the merging or splitting is specifically carried out when applied to a classification problem with K classes, to be considered in Section 16.2. The resulting binary tree, typically upside down, has $2K-1$ nodes, including K leaf nodes each for one of the classes, and $K-1$ non-leaf nodes, at each of which two subgroups are either merged into one group in the bottom-up case, or group is split into two subgroups in the top-down case.

Either of the top-down splitting and bottom-up merging methods can also be used to construct a hierarchical structure for unsupervised clustering if applied

to all N individual unlabeled data samples in the given dataset, instead of K labeled classes, as we will see in Section 16.3.

- **Bottom-Up method:**

  Initially, every class composed of data points with the same class identity is treated as subgroup represented by one of the K leaf nodes at the bottom of the binary tree. These subgroups are then merged two at a time to form progressively larger groups. This process is carried out recursively $K-1$ times until eventually all K classes are merged into a single super group at the root node of the tree. In the following discussion, we assume the dataset contains in total N data samples of which n_k are labeled to belong to class C_k, $(k = 1, \cdots, K)$. The steps for this bottom-up merging method is shown below.

  1.Compute the $K(K-1)/2$ pairwise Bhattacharyya distances between every two classes C_i and C_j:

$$d_B(C_i, C_j) = \frac{1}{4}(\mathbf{m}_i - \mathbf{m}_j)^T \left[\frac{\mathbf{\Sigma}_i + \mathbf{\Sigma}_j}{2}\right]^{-1} (\mathbf{m}_i - \mathbf{m}_j) + \log\left[\frac{\left|\frac{\mathbf{\Sigma}_i + \mathbf{\Sigma}_j}{2}\right|}{(|\mathbf{\Sigma}_i|\ |\mathbf{\Sigma}_j|)^{1/2}}\right] \tag{16.1}$$

  2.Merge the two classes corresponding to the smallest d_B to form a new class $C_k = C_i \cup C_j$, and compute its mean

$$\mathbf{m}_k = \frac{1}{n_i + n_j}[n_i\mathbf{m}_i + n_j\mathbf{m}_j] \tag{16.2}$$

  and covariance

$$\mathbf{\Sigma}_k = \frac{1}{n_i + n_j}[n_i(\Sigma_i + (\mathbf{m}_i - \mathbf{m}_k)(\mathbf{m}_i - \mathbf{m}_k)^T) + n_j(\Sigma_j + (\mathbf{m}_j - \mathbf{m}_k)(\mathbf{m}_j - \mathbf{m}_k)^T)] \tag{16.3}$$

  Delete the old classes C_i and C_j. Now there are $K-1$ classes left.

  3.Repeat the previous steps to find all pairwise distances for the remaining classes, and merge the two with minimum distance, until eventually all classes are merged into a single group containing all K classes, the binary tree structure is thus obtained.

- **Top-Down method:**

  Initially, all K classes are treated as members of a single super group at the root node on top of the tree, which is then split into two subgroups each represented as a node at a lower level of the tree. This subdivision is carried out recursively until eventually each subgroup contains a single class represented by one of the leaf nodes at the bottom of the tree. This tree structure can be constructed in either breadth-first manner based on a queue, or depth-first manner based on a stack, which can be most conveniently implemented recursively.

  Various methods can be used to subdivide a group into two subgroups. For example, the K-means method can be used with $K = 2$. Alternatively,

the binary subdivision can be carried out in the 1-D space along the direction of the principal component based on the KLT method, applied to either the covariance matrix of the entire dataset, or the between-class scatter matrix (Eq. (9.20)). All samoples are projected onto this direction and then subdivided into two parts with maximum Bhattacharyya distance, as shown in the following steps.

1. Compute the covariance matrix $\mathbf{\Sigma}$ of all N data points or thebetween-class scatter matrix $\mathbf{S}_B$ of all K classes, find its maximum eigenvalue λ_i and the corresponding eigenvectors $\mathbf{v}_i$;
2. Project all data points onto $\mathbf{v}_1$:

$$y_n = \mathbf{x}_n^T \mathbf{v} \qquad (n = 1, \cdots, N) \tag{16.4}$$

3. Sort all data points $\{y_1, \cdots, y_N\}$ along this 1-D space and partition them into two subgroups with maximum Bhattacharyya distance.
4. Carry out the steps above recursively to each of the two subgroups, until eventually every subgroup contains only one class.

The hierarchical structure so constructed by either method can then be used as either a decision tree classifier for supervised classification or unsupervised clustering as respectively discussed in the following two sections.

16.2 Binary Hierarchical Classification

The hierarchical classification method is especially suitable when the dimension of the feature space is high and the number of classes is large. For a single-level classifier, it may be difficult to select a subset of the large number of available features good for separating all classes in a single step. On the other hand, for a hierarchical classifier it is possible to select a small subset of features that are most relevant and suitable to distinguish two subgroups at each node. Although this is a multi-step process, it can be carried out effectively and efficiently due to the low computational complexity of the binary classification.

Specifically, any unlabeled sample to be classified enters the tree structure at the root node, where it is classified into either the left or right subgroup, and this binary classification is further carried out recursively at each of the tree nodes along a certain branch of the tree, until eventually reaching one of the K leaf nodes representing class identity of the data sample. Here are the specific steps:

1. According to the specific classification method used, find the discriminant functions $D_l(\mathbf{x})$ and $D_r(\mathbf{x})$ for the two subgroups based on the training data.
2. Select the best $d < D$ features most suitable for separating the two groups G_l and G_r, based on any of the feature selection methods such as those listed below:
   - Choosing d features directly from the D original ones using between-class distance (e.g., Bhattacharrya distance) as the criterion,

- Carry out KLT based on the between-class scatter matrix $\mathbf{S}_B$ and use the first d principal components for the binary classification.

As here only two groups of classes need to be distinguished, the number of selected features can be expected to be small.

3. Any unlabeled pattern $\mathbf{x}$ enters the classifier at the root of the tree and is classified to either the left or the right sub-group of the node according to the discriminant function

$$\mathbf{x} \in \begin{cases} G_l & if \;\; D_l(\mathbf{x}) > D_r(\mathbf{x}) \\ G_r & if \;\; D_r(\mathbf{x}) < D_l(\mathbf{x}) \end{cases} \tag{16.5}$$

This process is carried out recursively at each of the subsequent nodes until eventually $\mathbf{x}$ reaches one of the leaf nodes corresponding to a single class, to which the sample $\mathbf{x}$ is therefore classified.

The Matlab function listed below shows how the top-down splitting method can be implemented recursively as a depth-first search process. Here `X` is a 2-D data array containing N column vectors each for one of the N data samples labeled as one of the K classes, `M` is a 2-D array containg K columns each for the mean vector of one of the K classes, and `Cov` is a 3-D array containing the covariance matrices of the K classes (both pre-calculated in the main function). Also, `p` is a 1-D vector for the parent node of each of the $2K-1$ tree nodes, to be used for visualizing the tree using the Matlab function `treeplot`.

```
function TDclassify()          % top-down tree construction
    global X M Cov K p nc
    [d N]=size(X);             % dimension and size of data
    gid=1:K;                   % list of all K classes
    nc=K;                      % initialize current node
    p=zeros(1,2*K-1);          % parent list for all tree nodes;
    f=0;                       % parent node is always 0
    DFdivideC(gid,f);          % top-down binary tree construction
    treeplot(p)                % plot the binary tree
    [x,y]=treelayout(p);
    text(x+0.02,y, compose('%d',[1:2*K-1])) % label all tree nodes
end

function DFdivideC(gid,f)      % depth-first search recursion
    global X K M Cov p nc      % global variables shared by all functions
    if length(gid)==1          % leaf-node, terminate recursion
         return
    end
    nc=nc+1;                   % current number of nodes
    p(nc)=f;                   % assign parent
    f=nc;                      % parent node for its children
    [lid rid]=twoMeansC(gid); % split gid into two parts lid and rid
```

```
    if length(lid)==1          % left branch is a leaf node
        p(lid)=nc;
        if length(rid)==1      % leaf node on right
            p(rid)=nc;
        end
    else                       % left branch is non-leaf node
        DFdivideC(lid,f)       % recursion of the left branch
    end
    DFdivideC(rid,f)           % recursion of the right branch
end
```

The function below subdivides the given list of groups in `gid` into two subgroups `lid` on the left and `rid` on the right, based on the method of K-means clustering with $K = 2$.

```
function [lid rid]=twoMeansC(gid) % 2-means clustering
    global X M Cov Nk
    K=length(gid);           % number of classes in input
    if K==2
        lid=gid(1);
        rid=gid(2);
        return
    end
    m1=M(:,gid(1));          % first 2 classes as seeds for 2 means
    m2=M(:,gid(2));
    er=inf;
    while er>0               % clustering iteration
        gid1=[]; cid1=[];    % group IDs and classes IDs for group 1
        gid2=[]; cid2=[];    % group IDs and classes IDs for group 2
        m1o=m1;  m2o=m2;     % old means
        for l=1:K            % for each of K classes
            k=gid(l);        % kth cluster ID
            m=M(:,k);        % mean of cluster k
            c=Cov(:,:,k);    % covariance of cluster k
            d1=distM(m1o,m,c);
            d2=distM(m2o,m,c);
            if d1 < d2
                gid1=[gid1 k];              % assign class k to group 1
                cid1=[cid1 cid(k,1:Nk(k))]; % assign samples to group 1
            else
                gid2=[gid2 k];
                cid2=[cid2 cid(k,1:Nk(k))];
            end
        end
        m1=mean(X(:,cid1)')'; % two new means
```

```
        m2=mean(X(:,cid2)')';
        er=norm(m1-m1o)+norm(m2-m2o);
    end
    lid=gid1; rid=gid2;
    if length(gid2)==1        % single node always on left branch
        lid=gid2; rid=gid1;
    end
end
```

The bottom-up method for the binary tree construction keeps merging two groups of classes with minimum Bhattacharyya distance recursively from the leaf nodes each representing a single class all the way up to the root node representing a super group containing all classes. This process can be implemented by the Matlab function listed below, where `Nk` is a list containing the number of samples in each of the K classes. The main loop is carried out $K-1$ times to keep merging two subgroups (to be deleted) with minimum distance into a group (to be created). After $K-1$ such merge operations, all K classes are merged into a single super group at the root node. Again, the parent nodes of the $2K-1$ tree nodes are stored in a list `p` for the visualization of the binary tree.

```
function BUtree()           % bottom-up tree construction
    global X K Nk M Cov p groups
    [d N]=size(X);          % number of classes
    cid=zeros(2*K,N);       % sample IDs in each cluster
    del=ones(1,2*K);        % deleted clusters
    p=zeros(1,2*K-1);       % parents list;
    Ck=zeros(1,2*K);        % number of samples per group
    Ck(1:K)=Nk;             % for K leaf nodes
    ik=0;
    for k=1:K
        cid(k,1:Ck(k))=ik+1:ik+Ck(k);
        ik=ik+Ck(k);
    end
    M=zeros(d,2*K);         % means for K leaf nodes and K-1 nonleaf nodes
    Cov=zeros(d,d,2*K);     % covariance of all 2K-1 nodes
    k=K;
    while k<2*K-1           % merge K-1 times
        for l=1:k           % get mean and cov for each group
            M(:,l)=mean(X(:,cid(l,1:Ck(l)))')';
            Cov(:,:,l)=cov(X(:,cid(l,1:Ck(l)))');
        end
        dmin=inf;
        for k1=2:k
            if del(k1)          % if first group is not deleted
                for k2=1:k1-1
```

```
                    if del(k2)  % if second group is not deleted
                        d=distB(M(:,k1),Cov(:,:,k1),M(:,k2),Cov(:,:,k2));
                        if dmin>d
                            dmin=d; i=k1; j=k2; % save min-distance so far
                        end
                    end
                end
            end
        end
        k=k+1;
        fprintf('(%2d,%2d) --> %2d\n',i,j,k)
        p(i)=k; p(j)=k;         % assign parent Ck to Ci and Cj
        cid(k,1:Ck(i)+Ck(j))=[cid(i,1:Ck(i)) cid(j,1:Ck(j))];
        Ck(k)=Ck(i)+Ck(j);
        del(i)=0; del(j)=0;     % delete Ci and Cj
    end
end
```

Example 16.1 Fig. 16.1 shows the binary tree structures created by both bottom-up (left) and top-down (left) methods applied to the dataset containing 10 classes of handwritten digits 0 through 9, as shown at the bottom of the 10 leaf nodes in both cases.

In the top-down method, node $(K + 1) = 11$ is the root node from which the super group containing all K classes is split recursively into the 10 leaf nodes each for one of the 10 classes, while in the bottom-up method, the 10 leaf nodes are recursively merged into a super group containing all 10 classes at node $2K - 1 = 19$ as the root node. Comparing the trees constructed by the two methods, we see that they are similar in that all classes similar in shape are grouped together, e.g., digits 3, 5, and 8 form a subgroup at node 18 in the top-down tree and node 14 in the bottom-up tree, and digits 4, 7, and 9, form another subgroup at nodes 15 and node 13 in the respective trees, while digits 0, 2, and 6 with dissimilar shapes are separeted early in top-down method or merged into the groups late in bottom-up method. The minor difference between the two results is that digits 1 and 7 are considered to be similar in the top-down method, but not so in the bottom-up method.

The parent list for all $2K - 1 = 20 - 1 = 19$ tree nodes is also shown in Fig. 16.1 the tree structure generated by each of the two method. The 19 nodes are listed in the bottom row while their corresponding parent nodes are listed in the top row. For example, the parent node of leaf node 8 (for digit 7) is node 16 in the top-down method or node 12 in the bottom-up method. Based on this parent list, the tree struction can be automatically generated by the Matlab function `treeplot(p)`.

Once the binary tree structure is constructed, we can further construct a binary classifier at each of the $K-1 = 10-1 = 9$ non-leaf nodes based on a small set of

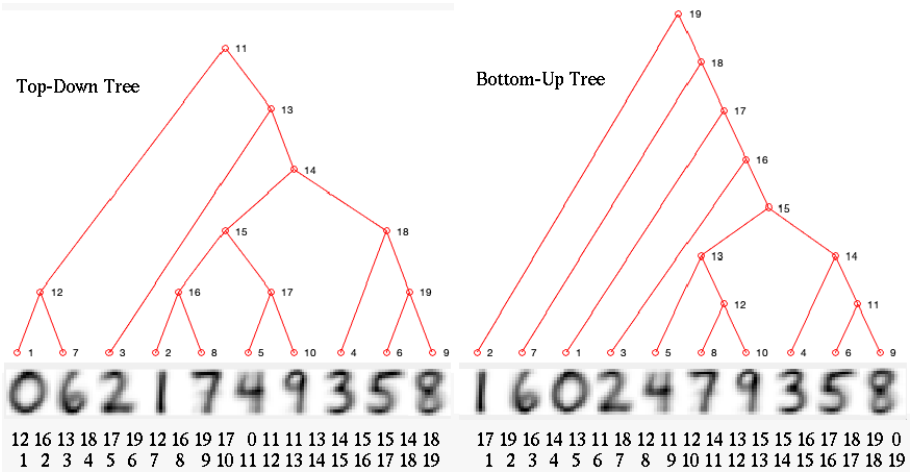


Figure 16.1 Binary Tree Classifier: Top-Down (left) and Bottom-Up (right)

features selected to saperate the two subgroups associated with the node. Such a sequence of binary classifications each based on a small set of features can be more effective and efficient than classifying all 10 class based on all features.

16.3 Binary Hierarchical Clustering

When the hierarchical method is applied to the clustering of a given dataset, the root node on top of the tree structure represents a super group containing all N data points in the dataset, while each of the leaf nodes is for one of these data point. A binary tree can be constructed in either top-down or bottom-up fashion. In the top-down method, the super group of all data points is recursively split into two subgroups in either depth-first or breadth-first fashion until reaching the bottom of the tree where each of the leaf nodes represents one of the N given data points. Alternatively, in the bottom-up method, starting from the leaf nodes each representing a single data sample, the two groups with minimum inter-group distance are merged to form a bigger group, and this process is further carried out recursively until reaching the top of the tree where the root node represents a super group containig all samples in the dataset.

The tree structure obtained by either method can then be truncated at certain level between the root node on top and the leaf nodes at the bottom to obtain a set of subgroups, each for one of the clusters, depending on the desired number and sizes of these clusters. Actually it is unnecessary to construct the complete tree first and then truncate it to get the clustering results. Instead, in either of the top-down or bottom-up methods, the recursion of splitting or merging can be terminated if certain conditions are met, such as the size of a group or the

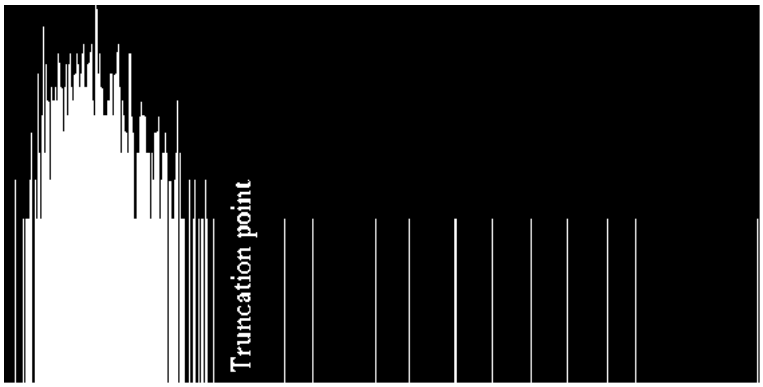


Figure 16.2 Histogram of Between-Subgroup Distances

intra-cluster scatteredness is too large or too small. By doing so, we obtain a set of clusters without constructing the complete tree.

Similar to all unsupervised clustering methods such as K-means method (Section 15.1), hierarchical clustering also faces the difficulty of determining the proper K value for the number of clusters that best reflects the inherent structure of the dataset. Here specifically we need to determine the parameter values (size of clusters, intra-cluster scatteredness, etc.) for terminating the recursive merging or splitting at a proper level. If the tree structure is truncated at a level which is either too high or too low, the resulting K may be either too large or too small, causing the dataset to be either under or over divided, as illustrated in Fig. 16.4 for an example to be considered later.

One way to address the issue of under versus over subdivision is to generate the histogram of all between-class distances at the tree nodes, as shown in Fig. 16.2. In general, the larger distances on the right side of the histogram come from the subdivisions at the top levels by which the dataset is divided into subgroups, and the smaller distances on the left are from the subdivisions at the lower levels by which some clusters in the dataset may be unnecessarily and over subdivided. In other words, the larger and smaller distances may reflect respectively the inter-cluster and intra-cluster scatteredness respectively. Also we note that in the histogram, there exists a dense peak composed of many small intra-group distances, much more than the number of larger inter-cluster distances coarsely distributed towards the right end of the histogram.

It is therefore clear that recursive subdivision should be truncated at certain point along the histogram that separates the valid larger inter-cluster distances on the right from the invalid smaller intra-cluster distances on the left, as also shown in Fig. 16.2. Based on this method the tree structure can be properly truncated as shown in Fig. 16.4 for Example 16.2, where a dataset is under (left), properly (middle), or over (right) clustered.

The Matlab code for both the depth-first and breadth-first algorithms is listed below, containing the following variables:

- `K`: the number of clusters (leaf nodes) so far (initially zero)
- `X`: a $d \times N$ array containing N d-dimensional column vectors each for one of the samples.
- `sid`: sample indices for all samples $(1, \cdots, N)$
- `kid`: cluster labelings $(1, \cdots, K)$ for all N samples, of which the nth component indicates the cluster sample $\mathbf{x}_n$ is assigned to.
- `path`: a list of paths (strings) for all N samples, only needed for visualization of the tree. For example, a sample with path '010' is assigned to the left, right, and left subgroups in three consecutive steps along a tree branch.
- `mins`: pre-specified minimum number of samples in a cluster.
- `th`: prespecified threshold value for within-cluster scatteredness.

The depth-first binary subdivision is based on based a stack (last-in-first-out), carried out recursively:

```
function DFdivide(sid)          % subdivide a node containg samples in sid
    global X K path kid th mins % global variables shared by other functions
    N=length(sid);              % number of samples
    Y=X(:,sid);                 % all samples at current node
    if cov2(Y)<th | N<mins      % terminate recursion (leaf node reached)
        K=K+1;                  % update number of clusters
        kid(sid)=K;             % assign cluster ID to all samples
        return
    end
% split group (sid) into left (lid) and right (rid) subgroups:
    [lid rid]=twoMeans(Y,sid);  % by either K-means (K=2)
%    [lid rid]=split(Y,sid);     % or splitting in principal direction
    for l=1:length(lid)         % for each sample on the left
        i=lid(l);               % index of each sample
        path{i}=[path{i},'0'];  % append 0 to path
    end
    for l=1:length(rid)         % for each sample on the right
        i=rid(l);               % index of each sample
        path{i}=[path{i},'1'];  % append 1 to path
    end
    % recursively subdivide the left and right nodes
    DFdivide(lid)
    DFdivide(rid)
end
```

Here `cov2` is a function that calculates the scatteredness, trace of covariance of column vectors in dataset `X`:

```
function tcov=cov2(X)
    [d,n]=size(X);
    tcov=0;
    if n>1
        tcov=trace(cov(X'));
    end
end
```

The breadth-first binary subdivision is based a queue (first-in-first-out), represented in the code by a dynamic 2-D array `Q`. Each tree node is represented by one row of `Q` containing indices of all samples assigned to the node (appended by zeros for all rows to have the same length).

```
function BFdivide(sid)
    global X K path kid th mins
    N=length(sid);                  % number of data samples
    Q=[[];sid];                     % initial queue: a row of all samples
    while size(Q,1)>0               % loop until the queue is empty
        q=Q(1,:);                   % get the first row
        sid=q(1:nnz(q));            % get sample IDs in the row
        Y=X(:,sid);                 % samples to be subdivided by either
%        [lid rid]=twoMeans(Y,sid);  % by K-means method (K=2)
        [lid rid]=split(Y,sid);     % or split along principal direction
        for l=1:length(lid)         % record splitting history
            i=lid(l);
            path{i}=[path{i},'0'];  % paths of samples in left branch
        end
        for l=1:length(rid)
            i=rid(l);
            path{i}=[path{i},'1'];  % paths of samples in right branch
        end
        if cov2(X(:,lid))<th | length(lid)<mins    % a leaf node
            K=K+1;                  % update number of clusters
            kid(lid)=K;             % assign cluster ID to all samples
        else                        % not a leaf node
            lid(end+1:N)=0;         % append zeros
            Q=[Q;lid];              % put left node back to queue
        end
        if cov2(X(:,rid))<th | length(rid)<mins  % same for left branch
            K=K+1;
            kid(rid)=K;
        else
            rid(end+1:N)=0;
            Q=[Q; rid];
        end
```

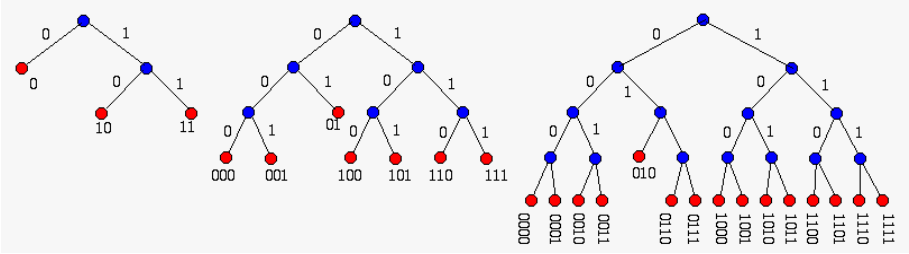


Figure 16.3 Tree Structures with 3, 7 and 15 Leaf Nodes Each for a Cluster

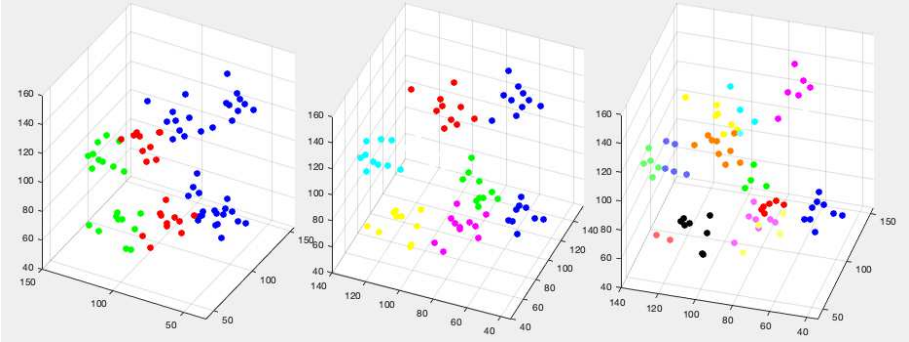


Figure 16.4 Under (left), Proper (middle), and Over (right) Division

```
        Q=Q(2:end,:);                % remove front row from queue
    end
end
```

Example 16.2 The top-down method is applied to the clustering of a set of 80 data points in a 4-D space, to generate a binary tree based on sequence of binary subdivision. The subdivision is terminated when either the number of samples or the intra-group scatteredness in a subgroup is smaller than a prespecified value. Based on three sets of such parameters, three tree structures are obtained as shown in Fig. 16.3, and the corresponding clustering results are shown in Fig. 16.4 (in 3-D space spanned by the first three pricipal components after KLT). We see that the dataset is best fit by the 7 clusters in the second case due to the properly selected parameters (`th`=0.01, `mins`=20), but it is either under divided (3 clusters in the first case) or over divided (15 clusters in the third case) due to improper parameters.

Example 16.3 The binary top-down splitting method is applied to the clustering analysis of a gene microarray containing a set of 99 genes (credit to the National Cancer Institute NIH, https://llmpp.nih.gov/lymphoma/data.shtml), in terms of their expression levels under seven different experimental conditions

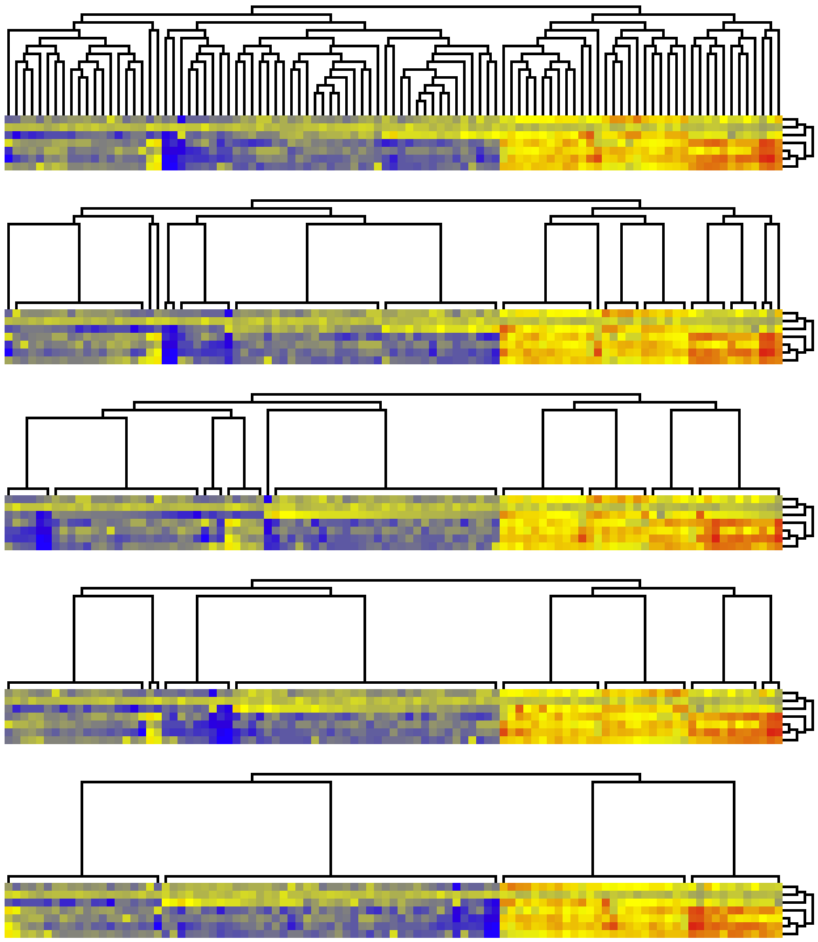


Figure 16.5 Clustering of 99 genes

as shown in Fig. 16.5 where the expression levels are color coded. These genes are clustered into various subsets of similar genes groups in terms their expression levels under different conditions. The five panels in the figure show how the full dendrogram (top) can be truncated at various levels to form different numbers of subsets. For example, the bottom panel shows that these genes can be clustered into four groups, which can be further merged into two larger groups representing very different behaviors under the seven conditions. The same method is also carried out for the seven experimental conditions (instead of the 99 genes) so that they can be also be clustered into the subgroups each containing a set of similar conditions, as shown by the smaller dendrogram on the right side of the microarray data.

The method is further applied to the clustering of a much larger microarray

dataset containing 13,411 genes (from the same source as above). The result is given in Fig. 16.6 showing only a small fragment of the complete dendrogram containing a small subset of the genes.

Problems

1. Develop your own code to implement a supervised hierarchical classifier by constructing a binary decision tree based on both top-down and bottom up methods, and then apply both of them to carry out classification of the hand-written digit dataset (10 classes with 256 features). Compare the two methods in terms of both accuracy (from the confusion matrices) and execution time.
2. Develop your own code for uusupervised hierarchical clustering based on both top-down and bottom-up methods, and apply them to the clustering of the hand-written digit dataset. Obtain the dendrograms and confusion matrices from the two methods and compare them in terms of accuracy and execution time.

   Extra credit: apply your clustering code to a microarray dataset of reasonable size downloaded from: https://llmpp.nih.gov/lymphoma/data.shtml.

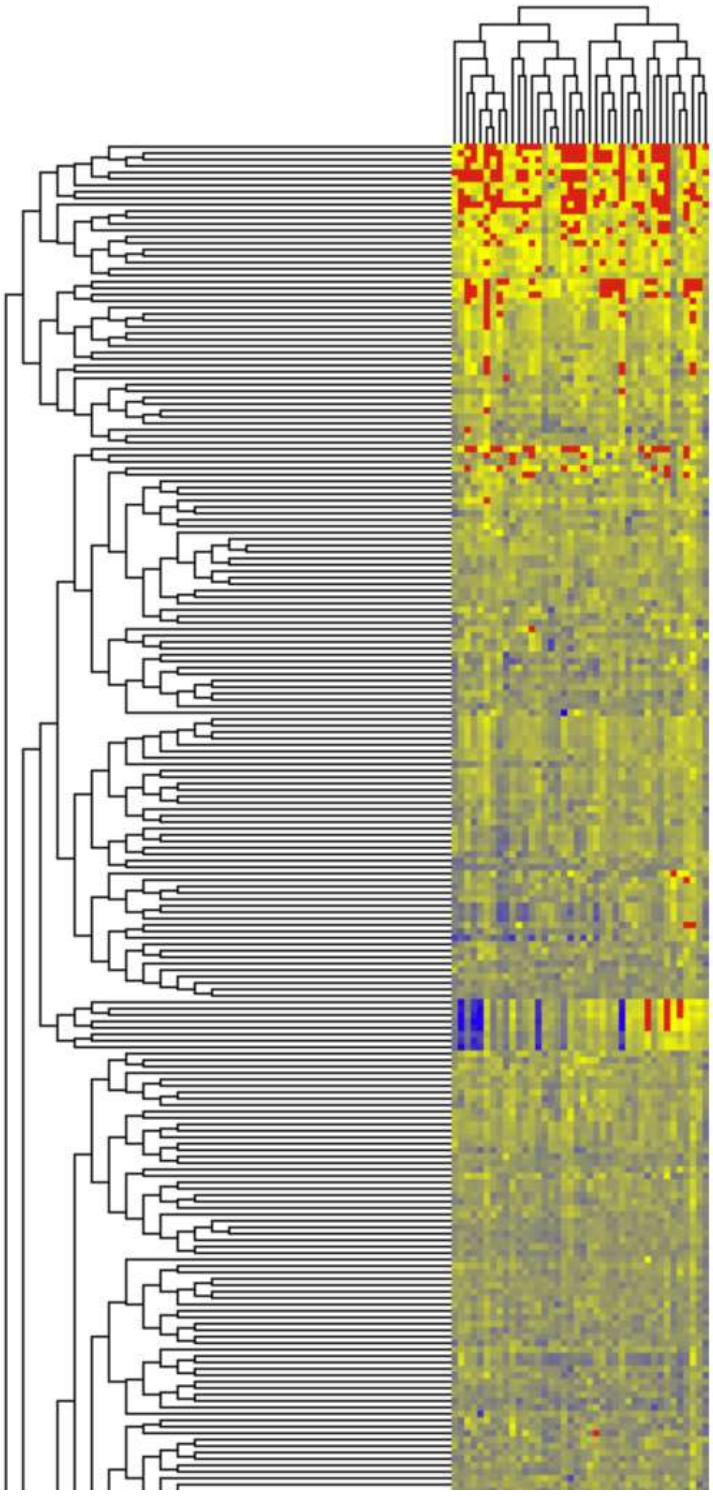


Figure 16.6 Clustering of 13,411 genes (only a small fragment shown)

Part V

Neural Networks

In Part V, we consider the general learning method of *artificial neural networks (ANN)*, a category of algorithms in machine learning that are inspired by the biological neural networks in the brain, composed of a large number of neural cells called neurons. An ANN is designed to carry out either supervised learning such as regression as well as classification, or unsupervised learning such as clustering. The ANN method is covered in Part V, separate from all supervised and unsupervised learning methods in the previous parts, simply because the ANN formulates the learning problems in a rather unique way (as we will see below), quite different from all previous methods. However, both supervised and unsupervised ANN algorithms can also be discussed together with all previously considered algorithms, as in the case of many other textbooks.

Before discussing specific learning algorithms and the actual computations taking place in an ANN, we could treat it as a black box that mimic the behavior of a certain generic system, which generates a set of outputs as the response to a set of inputs, in the sense that the output of the ANN is similar to that of the system, if both are given the same input, although the inner workings of the ANN may be completely different from that of the system. In other words, the ANN can be used to model a system only in terms of the relationship between its input and output.

Mimicking the biological neural networks, an ANN is typically a hierarchical structure composed of multiple layers of neurons, also called *nodes* in the context of ANN, including the input layer, the hidden layers, and the output layer. These layers are organized hierarchically in the sense that each layer takes input from a lower layer and generates output to feed into a higher layer. As the computations taking place through the multilayers are in general nonlinear, an ANN can be used to model complex behaviors of some sophisticated systems such as visual signal processing and object recognition in the brain.

To achieve a specific learning goal, an ANN needs to be iteratively trained to learn the desired behavior from the given dataset, so that its *weights* representing the connectivities between the nodes can be progressively modified from some random initial values to eventually produce the desired output given the input. In general, this learning can be either *supervised* or *unsupervised*.

In unsupervised learning, the samples in the dataset $\mathbf{X}$ are unlabeled, there exists no desired output for the ANN algorithm to aim at, and it needs to modify its weights on its own based on the internal structures in the dataset, to achieve the desired goal such as clustering, similar to the K-means method.

In supervised learning, each data sample $\mathbf{x}_n$ in $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ from the training set is labeled with the corresponding $\mathbf{y}_n$ in $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$, which serves as the target output (i.e., the desired output corresponding to the input $\mathbf{x}_n$), to be compared with the actual output $\hat{\mathbf{y}}_n$ generated by the network. Based the comparison, the weights are adjusted so that its future response to $\mathbf{x}_n$ will be closer to the desired $\mathbf{y}_n$, thereby reducing the error or residue $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}}$. Neural network algorithms formulated in this manner are based on the general learning

model illustrated in Fig. 4.1 in Section 4.1, applicable to both regression and classification tasks.

Among all supervised training algorithms, *backpropagation* is of fundamental importance. It is widely used not only as a stand-alone algorithm for tasks such as classification but also as a core learning component in various deep learning methods, including large-scale artificial intelligence systems such as the large language models discussed in Part VII.

17 Biologically Inspired Networks

17.1 Biological Neural Network Modeling

To understand how such a neural network algorithm works, we first consider some basic concepts in biological neural system. The human brain consists of 10^{11} *neurons* interconnected through about 10^{14} to 10^{15} *synaptic junctions* to form millions of neural networks for various functions. Hundreds specialized cortical areas are formed based on these networks for different information processing tasks.

Functionally, a neuron consists of the following three parts as shown in Fig. 17.1:

- The *cell body or soma* containing the nucleus of the cell;
- The *dendrites* that receive electrochemical stimulation (input impulses) from other neurons and propagate them to the cell body;
- The *axon:* that conducts the impulses (output) away from the cell body to other cells.

For the large number of neurons in a neural network to achieve a certain function collectively, they are connected by the *axo-dendritic synapses* where the dendrites of one neuron receive input stimuli transmitted through the axons of many other neuron. The computational model of such a neuron in the neural network, called a *node*, is also shown in Fig. 17.1, which receives input signals denoted by $x_1, \cdots, x_d$ from d other neurons and becomes activated to different extents in terms of its *action potential* or simply *activation*, as a weighted sum of the inputs:

$$a = \sum_{j=1}^{d} w_j x_j + b = \sum_{j=0}^{d} w_j x_j = \mathbf{w}^T \mathbf{x} \tag{17.1}$$

We note that this weighted sum is exactly the same as the linear regression model in Eq. (5.1) in Section 5. However, here the model parameters take on different meanings: b is the offset or bias, and w_j is a weight representing the synaptic connectivity to the jth input node:

$$w_j \begin{cases} > 0 & \text{excitatory input} \\ < 0 & \text{inhibitory input} \\ = 0 & \text{no connection} \end{cases} \tag{17.2}$$

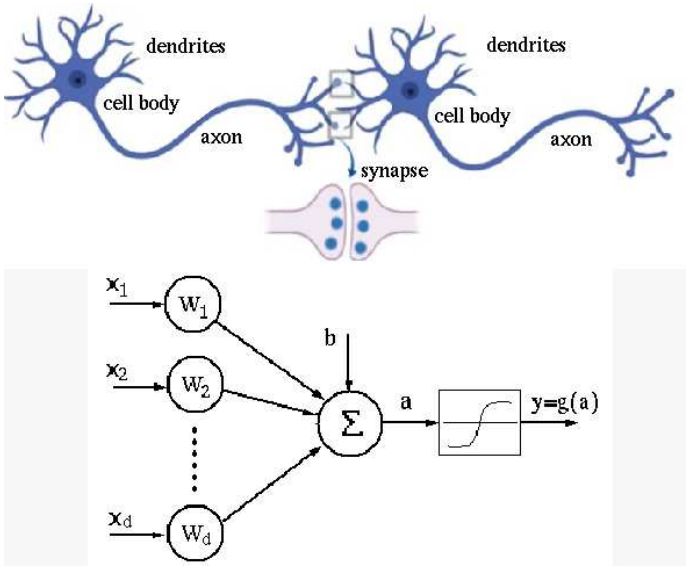


Figure 17.1 Computational Model of Neurons

Same as in linear regression, here we have defined $x_0 = 1$ and $w_0 = b$, so that both the weight and pattern vectors are augmented to become $d+1$ dimensional vectors $\mathbf{x} = [x_0 = 1, x_1, \cdots, x_d]^T$ and $\mathbf{w} = [w_0 = b, , w_1, \cdots, w_d]^T$. Based on the activation level a, the neuron produces an output or response as a function of a:

$$y = g(a) = g\left(\sum_{j=1}^{d} w_j x_j + b\right) = g(\mathbf{w}^T\mathbf{x}) \tag{17.3}$$

Here $g(x)$, the *activation function*, typically takes one of the following forms:

- Liner function:

$$g(x) = x, \qquad \frac{d\,g(x)}{dx} = 1 \tag{17.4}$$

- Logistic function:

$$g(x,a) = \frac{1}{1+e^{-ax}} = \frac{e^{ax}}{1+e^{ax}} = \begin{cases} 0 & x = -\infty \\ 1/2 & x = 0 \\ 1 & x = \infty \end{cases}, \qquad \frac{d\,g(x)}{dx} = \frac{a\,e^{-ax}}{(1+e^{-ax})^2} \tag{17.5}$$

where parameter a controls the slope of the transition from 0 to 1 around $x = 0$. Specially

$$g(x,0) = \frac{1}{2}, \qquad \frac{d\,g(x)}{dx} = 0 \tag{17.6}$$

$$g(x,\infty) = u(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}, \qquad \frac{d\,g(x)}{dx} = \begin{cases} 0 & x \neq 0 \\ \infty & x = 0 \end{cases} \tag{17.7}$$

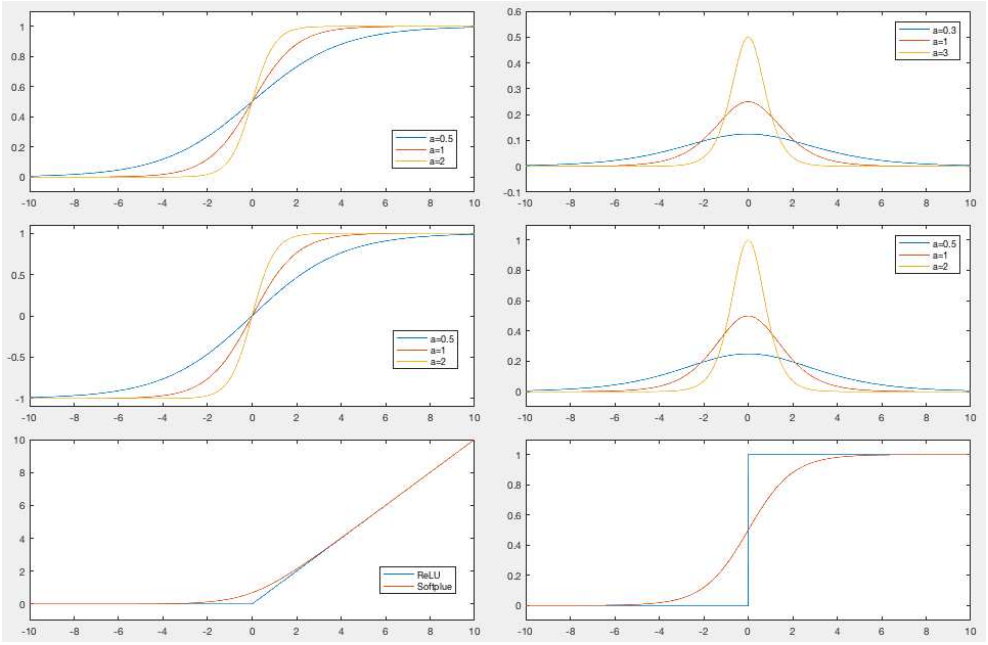


Figure 17.2 Activation Functions (left) and Their Derivatives (right) Logistic (top), Tanh (middle), and ReLU and Softplus (bottom)

- Tanh (hyperbolic tangent) function:

$$g(x,a) = \frac{2}{1+e^{-ax}} - 1 = \frac{e^{ax}-1}{e^{ax}+1} = \begin{cases} -1 & x = -\infty \\ 0 & x = 0 \\ 1 & x = \infty \end{cases}, \qquad \frac{d\,g(x)}{dx} = \frac{2a\,e^{-ax}}{(1+e^{-ax})^2} \tag{17.8}$$

- Rectified linear unit (ReLU):

$$g(x) = \max(0,\, x) = \begin{cases} 0 & x < 0 \\ x & x > 0 \end{cases}, \qquad \frac{d}{dx}g(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \tag{17.9}$$

  The soft version of ReLU is the softplus function of which the derivative is the logistic function:

$$g(x) = \log(1+e^{x}), \qquad \frac{d\,g(x)}{dx} = \frac{e^{x}}{1+e^{x}} = \frac{1}{1+e^{-x}} \tag{17.10}$$

These activation functions and their derivatives are shown in Fig. 17.2.

A neural network can be typically modeled by a hierarchical structure as shown in Fig. 17.3 containing two or more layers of neurons, called *nodes* in the ontext of artificial neural networks:

- The *input layer*: receives inputs from external sources;
- The *output layer*: generates output to the external world;

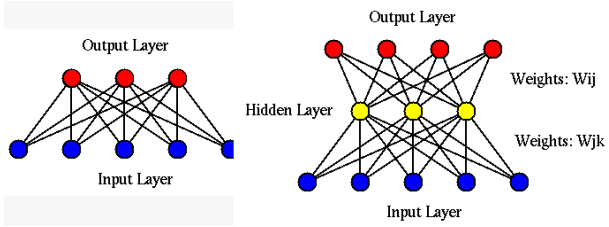


Figure 17.3 Artificial Neural Networks of Two (left) or Three (right) Layers

- The *hidden layer(s)*: one or more layers between the input and output layers, not visible from outside the network.

A neural network can be trained according to certain mathematical rules, the *learning rules* or *learning laws*, by modifying its weights iteratively based on the inputs (and the desired outputs if the learning is supervised), so that the network can produce the desired output as the response to its input as the stimulus.

The method of ANN can be used for both supervised and unsupervised learning. For supervised learning, the goal is to model the relationship between the dependent variable $\mathbf{y}$ as the output of the network and the independent variable $\mathbf{x}$ as the input of the network by a function $\mathbf{y} = f(\mathbf{x}, \mathbf{w})$, of which the parameter $\mathbf{w}$ for all weights in the network is to be estimated in the learning process based on the training set $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$. Such a general framework is suitable for both regression and classification. Neural networks can also be used for unsupervised learning based on observed data $\mathbf{X}$ without the labeling $\mathbf{y}$.

Listed below are several learning paradigms for various neural network algorithms with different interpretations of the input and output of a neural network.

1. *Pattern Associator*

   This is the most general form of neural network learning by which the associative relationship between two sets of patterns in $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ and $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$ is learned and stored.

   - Training: A set of N pairs of patterns $\{(\mathbf{x}_n, \mathbf{y}_n),\ n = 1, \cdots, N\}$ is presented to the network which then learns to establish the associative relationship between two sets of patterns:

$$f : \mathbf{x} \in \mathcal{R}^d \Longrightarrow \mathbf{y} \in \mathcal{R}^m \tag{17.11}$$

   - Testing: When an unlabeled pattern $\mathbf{x}$ in a pair is presented as the input, the network produces an output pattern $\mathbf{y}$ associated to the output.

   Human memory is associative in the sense that given one pattern, some associated pattern(s) may be produced. Examples include: (Evolution, Darwin), (Einstein, $E = mc^2$), (food, sounding bell, salivation).

2. *Auto-associator*

   As a special pattern associator, auto-associator associates a pre-stored pattern to an incomplete or noisy version of the pattern.

- Training: A set of patterns $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ is presented to the network for it to learn and remember, i.e., the patterns are stored in the network.
- Testing: When an incomplete or noisy version of one of the patterns stored in the network is presented as the input to the network, the original pattern is retrieved by the network as the output.

3. *Regression*

   This is another special kind of pattern associator which takes a vector input $\mathbf{x} \in \mathcal{R}^d$ and produces a real value $y \in \mathcal{R}$ as a multivariable function $y = f(\mathbf{x})$ at its only output node.
   - Training: trained by a set of observed data samples, the independent vectors and their corresponding function values $\{(\mathbf{x}_n,\ y_n),\ n = 1, \cdots, N\}$, the network is model the function.
   - Testing: given any vector $\mathbf{x}$, the output value produced by the single output node is an estimated function value $y = f(\mathbf{x})$.

4. *Classification*

   This is a variation of the pattern associator of which the output patterns are a set of categorical symbols representing different classes $\{C_1, \cdots, C_K\}$, i.e., each input pattern is classified by the network into one of the classes

$$f : \mathbf{x} \in \mathcal{R}^d \Longrightarrow y \in \{C_1, \cdots, C_K\} \tag{17.12}$$

5. *Regularity Detector*

   This is an unsupervised learning process. The network discovers automatically the regularity in the inputs so that similar patterns are automatically detected and grouped together in the same cluster or class.

17.2 Hebbian Learning

Donald Hebb speculated in his book *The Organization of Behavior: A Neuropsychological Theory* (1949) that

"When neuron A repeatedly and persistently takes part in exciting neuron B, the synaptic connection from A to B will be strengthened."

In other words, simultaneous activation of neurons leads to pronounced increases in synaptic strength between them, or "neurons that fire together wire together; neurons that fire out of sync, fail to link".

For example, the well known *classical conditioning* (Pavlov, 1927) could be explained by Hebbian learning. Consider the following three patterns:

- Unconditioned stimulus: sight of food F
- Conditioned stimulus: sound of bell B
- Response: secretion of saliva S

The unconditioned response is: $F \rightarrow S$. Due to the repeated and persistent conditioning process $F \cap B \rightarrow S$, the synaptic connections between patterns B and S are strengthened as both are repeatedly excited simultaneously, i.e., the two patterns become associated, resulting the conditioned response $B \rightarrow S$.

Based on this theory of Hebbian learning, the Hebbian network can be considered as a supervised learning method that learns to establish the associative relationship between any pair of two patterns $\mathbf{x}_n$ and $\mathbf{y}_n$ in the given dataset $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ and $\{\mathbf{y}_1, \cdots, \mathbf{y}_N\}$, considered as the training set. This is a 2-layer network with d nodes in the input layer to receive an input pattern $\mathbf{x} = [x_1, \cdots, x_d]^T$ and m nodes of the output layer to produce an output $\mathbf{y} = [y_1, \cdots, y_m]^T$. Each output node is fully connected to all d input nodes through its weights and generates an output:

$$y_i = \sum_{j=1}^{d} w_{ij} x_j = \mathbf{w}_i^T \mathbf{x} \quad (i = 1, \cdots, m) \tag{17.13}$$

where $\mathbf{w}_i = [w_{i1}, \cdots, w_{id}]^T$ is the weight vector for the ith output node. These m equations can be written in matrix form as

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^T \\ \vdots \\ \mathbf{w}_m^T \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} = \mathbf{W}\mathbf{x} \tag{17.14}$$

where $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_m]^T$ is an $m \times d$ matrix.

The Hebbian learning rule is inspired by Hebb's theory, i.e., when both neurons x_j and y_i are activated, the synaptic connectivity, here the weight w_{ij}, between them is enhanced:

$$w_{ij}^{new} = w_{ij}^{old} + \eta \, x_j y_i \quad (i = 1, \cdots, m, \; j = 1, \cdots, d) \tag{17.15}$$

or in matrix form:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \eta \, \mathbf{y}\mathbf{x}^T \tag{17.16}$$

Here η is the *learning rate*, a parameter that controls how quickly the weights get modified.

As in all supervised learning, the Hebbian network is first trained and then used for association.

- **Training:**

  For simplicity, we assume all weights are initialized to zero $w_{ij} = 0 \quad (i = 1, \cdots, m, \; j = 1, \cdots, d)$, and then train the network to find all weights based on all N pattern pairs in the dataset $\{(\mathbf{x}_n, \mathbf{y}_n), \quad n = 1, \cdots, N\}$ based on the learning law:

$$w_{ij} = \sum_{n=1}^{N} x_j^{(n)} y_i^{(n)} \quad (i = 1, \cdots, m, \; j = 1, \cdots, d) \tag{17.17}$$

or in matrix form, the weight matrix is the sum of the outer-products of all N pairs of patterns:

$$\mathbf{W}_{m\times d} = \sum_{n=1}^{N} \mathbf{y}_n \mathbf{x}_n^T = \sum_{n=1}^{N} \begin{bmatrix} y_1^{(n)} \\ \vdots \\ y_m^{(n)} \end{bmatrix} [x_1^{(n)}, \cdots, x_d^{(n)}] \tag{17.18}$$

- **Association:**

  When one of the patterns $\mathbf{x}_l$ is presented to the network, it produces an output:

$$\mathbf{y} = \mathbf{W}\mathbf{x}_l \tag{17.19}$$

  which, as shown below, is the same as the associated pattern $\mathbf{y}_l$, under the following conditions:

  – The N patterns $\{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ are treated as zero-mean random vectors, and they are assumed to be independent of each other with zero covariance and therefore zero *correlation coefficient* between any $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$r_{ij} = \frac{\sigma_{ij}^2}{\sigma_i \sigma_j} = \frac{\sum_{k=1}^{d} x_{ki} x_{kj}}{\sqrt{\sum_{k=1}^{d} x_{ki}^2} \sqrt{\sum_{k=1}^{d} x_{kj}^2}} = \frac{\mathbf{x}_i^T \mathbf{x}_j}{||\mathbf{x}_i||\,||\mathbf{x}_j||} = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{17.20}$$

    If we further assume all patterns are normalized with $||\mathbf{x}_n|| = 1$, then we have $\mathbf{x}_i^T \mathbf{x}_j = \delta_{ij}$

  – The number of input nodes is greater than the number of pattern pairs: $d \geq N$, i.e., the capacity of the network is large enough for representing N different patterns (as there can be no more than d orthogonal vectors in a d-dimensional space).

  Under these assumed conditions, the output of the network as its response to input $\mathbf{x}_l$ is

$$\mathbf{y} = \mathbf{W}\mathbf{x}_l = \left(\sum_{n=1}^{N} \mathbf{y}_n \mathbf{x}_n^T \right) \mathbf{x}_l = \mathbf{y}_l (\mathbf{x}_l^T \mathbf{x}_l) + \sum_{n \neq l} \mathbf{y}_n (\mathbf{x}_n^T \mathbf{x}_l) = \mathbf{y}_l \tag{17.21}$$

  as $\mathbf{x}_n^T \mathbf{x}_l = 0$ for all other terms with $n \neq l$. We see that a one-to-one correspondence relationship between $\mathbf{x}_n$ and $\mathbf{y}_n$ has been established for all $n = 1, \cdots, N$.

  In non-ideal cases where the conditions above are not strickly satisfied, e.g., the patterns in the dataset are not completely independent, the summation term is non-zero and there will be a non-zero error $\mathbf{y} - \mathbf{y}_l \neq 0$. However, the Hebbian network trained in this way may still behave as an associator that mimic approximately the general associative relationship between $\mathbf{x}_l$ and $\mathbf{y}_l$.

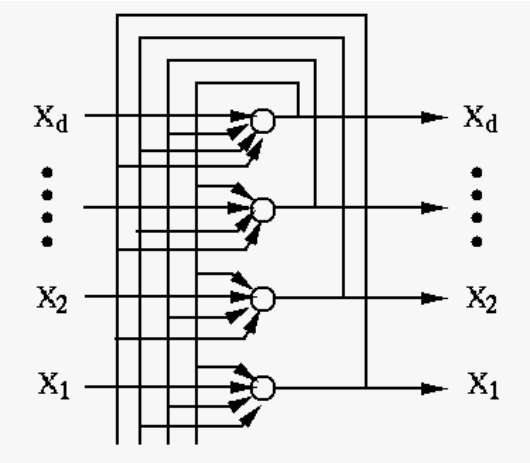


Figure 17.4 The Hopfield Network

17.3 Hopfield Network

The Hopfield network is a *recurrent neural network (RNN)* that allows the previous outputs of the network to be fed back and used as inputs again (with time delay) as an iterative process, so that it can learn to respond to inputs as a temporal sequence, such as a speech signal.

The structure of a Hopfield network is illustrated in Fig. 17.4. The learning of the Hopfield network is based on the Hebbian learning rule, and as a supervised method, it is trained on a dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_K]$ of K data samples each composed of d binary components $\mathbf{x} = [x_1, \cdots, x_d]^T$ with $x_i \in \{-1,\ 1\}$, $(i = 1, \cdots, d)$, representing one of K different patterns of interest, used as the inputs to the d nodes of network. Once the network is completely trained, the weight matrix $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_d]$ of the network is obtained in which the K patterns are stored. Given an input $\mathbf{x}$, an iterative computation is carried out until convergence, when one of the pre-stored pattern that most closely resemble the current input $\mathbf{x}$ is produced as the output. The Hopfield network can therefore be considered as an auto-associator (a content addressable memory), by which a pre-stored pattern can be retrieved based on its association to the input pattern established by training process of the network.

- **Training:**

  The training process is essentially the same as the Hebbian learning, except here the two associated patterns in each pair are the same (self-association). The weight matrix of the network is obtained as the sum of

the outer-products of the K patterns to be stored:

$$\mathbf{W}_{d\times d} = \sum_{k=1}^{K} \mathbf{x}_k \mathbf{x}_k^T = \sum_{k=1}^{K} \begin{bmatrix} x_1^{(k)} \\ \vdots \\ x_d^{(k)} \end{bmatrix} [x_1^{(k)}, \cdots, x_d^{(k)}] \tag{17.22}$$

The weight connecting node i and node j is defined as

$$w_{ij} = \sum_{k=1}^{K} x_i^{(k)} x_j^{(k)} = w_{ji} \tag{17.23}$$

We assume no self-connection exists $w_{ii} = 0$ $(i = 1, \cdots, d)$

- **Autoassociation:**

  Once the weight matrix $\mathbf{W}$ is obtained by the training process, the network can be used as a self-associator. When an input pattern $\mathbf{x}$ is presented to the network, the outputs of the network are updated *iteratively* and *asynchronously*, one randomly selected node at a time:

$$x_i^{(n+1)} = \text{sign}\left(\sum_{j=1}^{d} w_{ij} x_j^{(n)}\right) = \begin{cases} +1, & \text{if } \sum_{j=1}^{d} w_{ij} x_j^{(n)} \geq 0 \\ -1, & \text{if } \sum_{j=1}^{d} w_{ij} x_j^{(n)} < 0 \end{cases} \tag{17.24}$$

  where $x_i^{(n)}$ and $x_i^{(n+1)}$ are the output x_i of the ith node before and after the nth iteration, respectively. As shown below, this iteration will always converge to one of the K pre-stored patterns.

We first define the *Energy function* of any two nodes x_i and x_j of $\mathbf{x}$ as

$$e_{ij} = -w_{ij} x_i x_j \tag{17.25}$$

and the total *energy* of all d nodes in the network as the sum of all pair-wise energies:

$$\mathcal{E}(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{d} e_{ij} = -\frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{d} w_{ij} x_i x_j = -\frac{1}{2} \mathbf{x}^T \mathbf{W} \mathbf{x} \tag{17.26}$$

The interaction between these two nodes is summarized below:

	x_j	x_i	$w_{ij} > 0$	$w_{ij} < 0$
1	-1	-1	$e_{ij} < 0$	$e_{ij} > 0$
2	-1	1	$e_{ij} > 0$	$e_{ij} < 0$
3	1	-1	$e_{ij} > 0$	$e_{ij} < 0$
4	1	1	$e_{ij} < 0$	$e_{ij} > 0$

(17.27)

We make two observations.

- When the two nodes i and j reinforce each other's state, $e_{ij} < 0$:
  - If $w_{ij} > 0$ (in cases 1 and 4), $x_j = \mp 1$ tends to keep x_i to stay at the same state ∓ 1 in the iteration.

  - If $w_{ij} < 0$ (in cases 2 and 3), $x_j = \mp 1$ tends to keep x_i to stay at the same state ± 1.
- When the two nodes i and j change each other's state, $e_{ij} > 0$:
  - If $w_{ij} < 0$ (in cases 1 and 4), $x_j = \mp 1$ tends to reverse x_i from its previous state ∓ 1 to ± 1.
  - If $w_{ij} > 0$ (in cases 2 and 3), $x_j = \mp 1$ tends to reverse x_i from its previous state ± 1 to ∓ 1.

We note that low energy e_{ij} corresponds to a stable interaction between x_i and x_j, i.e., they tend to remain unchanged, and high energy corresponds to an unstable interaction, i.e., they tend to change their states. As the result, low total $\mathcal{E}(\mathbf{x})$ corresponds to more stable condition of the network, while high $\mathcal{E}(\mathbf{x})$ corresponds to less stable condition.

We further show that the total energy $\mathcal{E}(\mathbf{x})$ always decreases whenever the state of any node changes. Assume x_k has just been changed, i.e.,$x_k^{(n+1)} \neq x_k^{(n)}$ ($x_k = \pm 1$ but $x_k^{(n+1)} = \mp 1$), while all others remain the same $x_{l\neq k}^{(n+1)} = x_{l\neq k}^{(n)}$. The energy before x_k changes state is

$$\begin{aligned}\mathcal{E}^{(n)}(\mathbf{x}) &= -\frac{1}{2}\left[\sum_{i\neq k}\sum_{j\neq k} w_{ij}x_i^{(n)}x_j^{(n)} + \sum_i w_{ik}x_i^{(n)}x_k^{(n)} + \sum_j w_{kj}x_k^{(n)}x_j^{(n)}\right] \\ &= -\frac{1}{2}\sum_{i\neq k}\sum_{j\neq k} w_{ij}x_i x_j - \sum_i w_{ik}x_i^{(n)}x_k^{(n)}\end{aligned}$$

and the energy after x_k changes state is

$$\mathcal{E}^{(n+1)}(\mathbf{x}) = -\frac{1}{2}\sum_{i\neq k}\sum_{j\neq k} w_{ij}x_i x_j - \sum_i w_{ik}x_i^{(n+1)}x_k^{(n+1)} \tag{17.28}$$

The energy difference is

$$\triangle\mathcal{E} = (x_k^{(n)} - x_k^{(n+1)})\sum_i w_{ik}x_i \tag{17.29}$$

Consider two cases:

- Case 1: if $x_k^{(n)} = -1$, but $\sum_i w_{ik}x_i \geq 0$ and $x_k^{(n+1)} = 1$, we have $(x_k^{(n)} - x_k^{(n+1)}) = -2 \leq 0$ and $\triangle\mathcal{E} \leq 0$.
- Case 2: if $x_k^{(n)} = 1$, but $\sum_i w_{ik}x_i < 0$ and $x_k^{(n+1)} = -1$, we have $(x_k^{(n)} - x_k^{(n+1)}) = 2 \geq 0$ and $\triangle\mathcal{E} \leq 0$.

As in either case, $\triangle\mathcal{E}(\mathbf{x}) \leq 0$ is always true throughout the iteration, the energy $\mathcal{E}(\mathbf{x})$ will eventually reach one of the minima of the *energy landscape* shown in Fig. 17.5, i.e., the iteration will always converge.

We further show that each of the K pre-stored patterns corresponds to one of the minima of the energy function. The energy function $\mathcal{E}(\mathbf{x})$ corresponding to

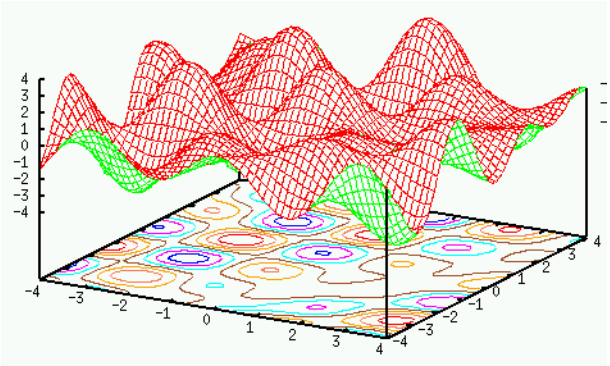


Figure 17.5 Energy Landscape

any $\mathbf{x}$ can be written as

$$\mathcal{E}(\mathbf{x}) = -\frac{1}{2}\mathbf{x}^T\mathbf{W}\mathbf{x} = -\frac{1}{2}\mathbf{x}^T\left[\sum_{k=1}^{K}\mathbf{x}_k\mathbf{x}_k^T\right]\mathbf{x} = -\frac{1}{2}\sum_{k=1}^{K}\mathbf{x}^T\mathbf{x}_k\mathbf{x}_k^T\mathbf{x} = -\frac{1}{2}\sum_{k=1}^{K}(\mathbf{x}^T\mathbf{x}_k)^2 \tag{17.30}$$

If $\mathbf{x}$ is different from (ideally orthogonal to) any of the K stored patterns, all terms in the summation will be small (ideally zero). But if $\mathbf{x}$ is similar to (ideally the same as) any one of the stored patterns, their inner product will be large (ideally reach maximum), causing the total energy to be minimized to reach one of the minima. In other words, the patterns stored in the network correspond to the local minima of the energy function. i.e., these patterns become *attractors*.

Note that it is possible to have other local minima, called *spurious states*, which do not represent any of the stored patterns, i.e., the associative memory is not perfect.

18 Perceptron-Based Networks

18.1 Perceptron

The perceptron network, first proposed by Frank Rosenblatt in 1958 ("The perceptron: a probabilistic model for information storage and organization", *Psychological Review* 65(6), 386-408), is a two-layer network containing an input layer of d nodes and an output layer of m output node. As a supervised method, it is trained by a dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, of which each sample $\mathbf{x} = [x_1, \cdots, x_d]^T$ is labeled by the corresponding component in $\mathbf{y} = [y_1, \cdots, y_N]^T$, indicating to which one of K categorical classes $\mathbf{x}_n$ belongs. The goal of the training process is to determine the weights $\mathbf{w} = [w_1, \cdots, w_d]^T$ and bias b so that the output $\hat{y}_n = \sum_{i=1}^{d} w_i x_n + b$ corresponding to input $\mathbf{x}_n$ matches its labeling y_n for all N samples in the training set in some optimal way. When the perceptron network is fully trained, it can be used to classify any unlabeled sample $\mathbf{x}$ into one of the K classes.

We first consider the special case where the output layer has only $m = 1$ node, and the perceptron is a binary classifier, same as the method of linear regression, and also the initial setup of the support vector machine (SVM), in the sense that all such methods are based on the linear equation $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$, representing a hyperplane by which the d-dimensional feature space is partitioned into two regions corresponding to two classes C_+ and C_-. When the parameters $\mathbf{w}$ and b are determined in the training process, any unlabeled $\mathbf{x}$ is classified into either of the two classes depending on whether the function value $\hat{y} = f(\mathbf{x})$ is greater or smaller than zero:

$$\text{If } f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b \begin{cases} > 0 \\ < 0 \end{cases}, \quad \text{then } \hat{y} = \text{sign}(f(\mathbf{x})) = \begin{cases} 1 & \text{for } \mathbf{x} \in C_+ \\ -1 & \text{for } \mathbf{x} \in C_- \end{cases} \tag{18.1}$$

If we divide the equation $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$ by $||\mathbf{w}||$, we get

$$p_{\mathbf{w}}(\mathbf{x}) = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||} = -\frac{b}{||\mathbf{w}||} = b' \tag{18.2}$$

where $p_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}/||\mathbf{w}||$ is the projection of $\mathbf{x}$ onto the normal direction $\mathbf{w}$ of the partitioning hyperplane, and $b' = -b/||\mathbf{w}||$ is the vector from the origin to the hyperplane ($|b'|$ is the distance of the hyperplane to the origin). Now the

classification above can be rewritten as

$$\text{If } p_{\mathbf{w}}(\mathbf{x}) \begin{cases} > b' \\ < b' \end{cases}, \quad \text{then} \quad \mathbf{x} \in \begin{cases} C_+ \\ C_- \end{cases} \tag{18.3}$$

We see that in all binary classification methods, an unlabeled sample $\mathbf{x}$ is classified into either of the two classes based on the projection $p_{\mathbf{w}}(\mathbf{x})$ of $\mathbf{x}$ onto $\mathbf{w}$, which is either greater or smaller than the bias b', indicating $\mathbf{x}$ is on either the positive or negative side of the plane.

In all these binary classification methods the parameters $\mathbf{w}$ and b need to be determined based on training set $\mathbf{X}$ labeled by $\mathbf{y}$, but they do so differently. While in least squares method and support vector machine these parameters are obtained by the least squared method and quadratic programming respectively, here in the perceptron network these parameters are obtained by iteratively modifying some randomly initialized values to gradually reduce the error or residual, the difference between actual output $\hat{y}_n = f(\mathbf{x}_n)$ and the ground truth labeling y_n, denoted by $\delta = y_n - \hat{y}_n$, for all training samples in the dataset. This method is called the δ-rule.

We now consider specifically the training algorithm of the perceptron network as a binary classifier. As always, we redefine the data vector as $\mathbf{x} = [x_0 = 1, x_1, \cdots, x_n]^T$ and the weight vector as $\mathbf{w} = [w_0 = b, w_1, \cdots, w_n]^T$ so that the decision function can be more conveniently written as an inner product $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ without the additional bias term.

The randomly initialized weight vector $\mathbf{w}$ is iteratively updated based on the following error driven δ-rule:

$$\mathbf{w}^{new} = \mathbf{w}^{old} + \eta(y - \hat{y})\,\mathbf{x} = \mathbf{w}^{old} + \eta\,\delta\mathbf{x} \tag{18.4}$$

where $\eta > 0$ is the step size but here called the *learning rate*, which is assumed to be 1 in the following for simplicity. We can show that by the iteration above, $\mathbf{w}$ is modified in such a way that the error δ is always reduced.

When a training sample $\mathbf{x}$ labeled by $y \pm 1$ is presented to the nodes of the input layer of the perceptron, its output $\hat{y} = \text{sign}(f(\mathbf{x}))$ may or may not match the the label y, as shown in the table:

	y	$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, $\hat{y} = \text{sign}(f(\mathbf{x}))$	$\delta = y - \hat{y}$
1	$y = 1$	$f(\mathbf{x}) > 0,\ \hat{y} = 1$	$\delta = 0$
2	$y = -1$	$f(\mathbf{x}) > 0,\ \hat{y} = 1$	$\delta = -2$
3	$y = 1$	$f(\mathbf{x}) < 0,\ \hat{y} = -1$	$\delta = 2$
4	$y = -1$	$f(\mathbf{x}) < 0,\ \hat{y} = -1$	$\delta = 0$

(18.5)

In both the first and last cases, $yf(\mathbf{x}) = 1 > 0$, the error is $\delta = y - \hat{y} = 0$, and the weight vector $\mathbf{w}^{new} = \mathbf{w}^{old} + \delta\mathbf{x} = \mathbf{w}^{old}$ is not modified. But in cases 2 and 3, $yf(\mathbf{x}) = -1 < 0$, the error is $\delta = y - \hat{y} \neq 0$, the weight vector $\mathbf{w}$ is to be modified in either of the following two ways:

- In case 2, $y = -1$, but $\hat{y} = 1$, then $\delta = -2$ and $y\,f(\mathbf{x}) < 0$, the weight vector

is update by

$$\mathbf{w}^{new} = \mathbf{w}^{old} - 2\mathbf{x} \tag{18.6}$$

When the same $\mathbf{x}$ is presented to the network again in the future, the function is

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{w}^{new} = \mathbf{x}^T\mathbf{w}^{old} - \eta 2\mathbf{x}^T\mathbf{x} < \mathbf{x}^T\mathbf{w}^{old} \tag{18.7}$$

which is smaller than its previous value and the output $\hat{y}$ is more likely to be the same as the desired $y = -1$.

- In case 3, $y = 1$, but $\hat{y} = -1$, then $\delta = 2$ and $y\,f(\mathbf{x}) < 0$, the weight vector is update by

$$\mathbf{w}^{new} = \mathbf{w}^{old} + 2\mathbf{x} \tag{18.8}$$

When the same $\mathbf{x}$ is presented again, the function is

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{w}^{new} = \mathbf{x}^T\mathbf{x}^{old} + \eta 2\mathbf{x}^T\mathbf{x} > \mathbf{x}^T\mathbf{w}^{old} \tag{18.9}$$

which is greater than its previous value and the output $\hat{y}$ is more likely to be the same as the desired $y = 1$.

The update equations in Eq. (18.6) for $y = -1$ and Eq. (18.8) for $y = -1$ can be combined to become

$$\mathbf{w}^{new} = \mathbf{w}^{old} + y\mathbf{x} \tag{18.10}$$

where the scaling constant 2 is dropped as it can be absorbed into the learning rate if we let $\eta = 1/2$. Now the learning rule can be rewritten as:

$$\text{If} \quad y\,f(\mathbf{x}) \begin{cases} > 0 \\ < 0 \end{cases} \quad \text{then} \quad \begin{cases} \mathbf{w}^{new} = \mathbf{w}^{old} \\ \mathbf{w}^{new} = \mathbf{w}^{old} + y\,\mathbf{x} \end{cases} \tag{18.11}$$

In summary, the learning law guarantees that the weight vector $\mathbf{w}$ is modified in such way that the performance of the network is always improved with reduced error $\delta = y - \hat{y}$. If C_+ and C_- are linearly separable, then a perceptron will always produce $\mathbf{w}$ in finite number of iterations to separate them.

This binary classifier with $m = 1$ output node can be generalized to multiclass classifier with $m > 1$ output nodes, and each of the m weight vectors in $\mathbf{W} = [\mathbf{w}_1, \cdots, \mathbf{w}_m]$ is modified by the same learning process considered above. The $K > 2$ classes can be encoded by the m outputs $y_i \in \{-1, 1\}$, $(i = 1, \cdots, m)$ in two different ways. If the *one-hot* method is used, the m binary output can encode $K = m$ classes, i.e., the kth class is represented by an m-bit output with $y_k = 1$ while all others $y_l = -1$ for $l \neq k$. Alternatively, if binary encoding is used, the m outputs can encode as many as $K = 2^m$ classes. For example, $K = 4$

classes can be labeled by 4 binary vector of either 4 or 2 bits:

binary output	$\mathbf{y}_0$	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$
y_1	1	−1	−1	−1
y_2	−1	1	−1	−1
y_3	−1	−1	1	−1
y_4	−1	−1	−1	1

or

binary output	$\mathbf{y}_0$	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$
y_1	−1	−1	1	1
y_2	−1	1	−1	1

(18.12)

The outputs of the m output nodes form an m-dimensional binary vector $\hat{\mathbf{y}} = [\hat{y}_1, \cdots, \hat{y}_m]^T$, which is to be compared with the labeling $\mathbf{y}$ for the current input $\mathbf{x}$ with error $\delta = ||\mathbf{y} - \hat{\mathbf{y}}||$. When the training is complete, any unlabeled input $\mathbf{x}$ is classified to one of the K classes with a matching label to the perceptron's output. In the case of one-hot encoding, it is possible for the binary output $\mathbf{y}$ to have more than one hot node (e.g., $\hat{\mathbf{y}} = [-1\ 1\ -1\ 1]^T$) not match any of the K one-hot encoded classes. In this case, the input $\mathbf{x}$ can be classified to the class corresponding to the node with the greatest output value $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$.

The Matlab code for the essential part of the algorithm is listed below. $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ is the dataset contains N training samples, each labeled by one of the components of $\mathbf{y} = [y_1, \cdots, y_n]$ as its class identity. $\mathbf{W}$ is a $(d+1) \times m$ matrix containing m $d+1$ dimensional weight vectors each for one of the m output nodes.

```
[X Y]=DataOneHot;       % get data
K=length(unique(Y','rows')) % number of classes
X=[ones(1,N); X];       % data augmentation
[d N]=size(X);          % numbers of dimensions and samples
m=size(Y,1);            % number of output nodes
W=2*rand(d,m)-1;        % random initialization of weights
eta=1;
nt=10^4;                % maximum number of iteration
for it=1:nt
    n=randi([1 N]);     % random index
    x=X(:,n);           % pick a training sample x
    y=Y(:,n);           % label of x
    yhat=sign(W'*x);    % binary output
    delta=y-yhat;       % error = desired - actual
    for i=1:m           % for each of m output nodes
        W(:,i)=W(:,i)+eta*delta(i)*x;
                        % update weights for ith output node
    end
    if ~mod(it,N)       % test for every epoch
        er=test(X,Y,W);
        if er<10^(-9)
            break
```

```
        end
    end
end
```

This is the function that tests the training set based on estimated weight vectors in $\mathbf{W}$:

```
function er=test(X,Y,W)  % test based on estimated W,
    [d N]=size(X);
    Ne=0;                % number of misclassifications
    for n=1:N
        x=X(:,n);
        yhat=sign(W'*x);
        delta=Y(:,n)-yhat;
        if any(delta)    % misclassification in some output nodes
            Ne=Ne+1;     % update number of misclassifications
        end
    end
    er=Ne/N;             % error percentage
end
```

This is the code that generates the training set labeled by either one-hot or binary encoding method:

```
function [X,Y]=Data
    d=3;  K=8;
    onehot=input('One-hot (1) or binary encoding (0): ');
    Means=[-1 -1 -1 -1 1 1 1 1;
           -1 -1 1 1 -1 -1 1 1;
           -1 1 -1 1 -1 1 -1 1];
    Nk=50*ones(1,K);
    N=sum(Nk);           % total number of samples
    X=[];
    Y=[];
    s=0.2;               % variance of noise
    for k=1:K            % for each of K classes
        Xk=Means(:,k)+s*randn(d,Nk(k));
        if onehot
            Yk=-ones(K,Nk(k));
            Yk(k,:)=1;
        else             % binary encoding
            dy=ceil(log2(K));
            y=2*de2bi(k-1,dy)-1;
            Yk=repmat(y',1,Nk(k));
         end
```

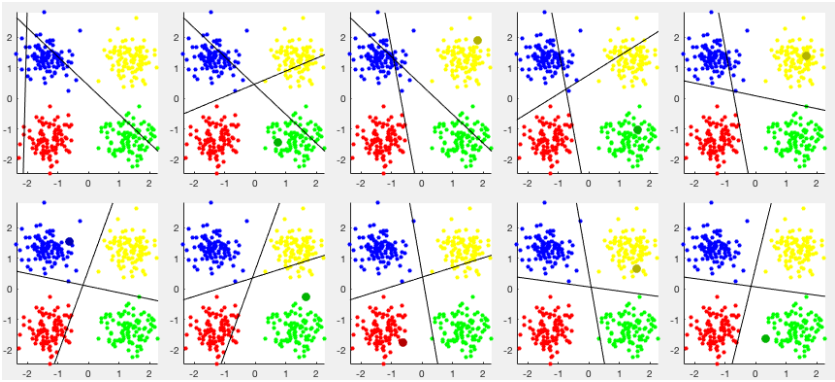


Figure 18.1 Classification by Perceptron (2-D)

```
        X=[X Xk];
        Y=[Y Yk];
    end
    Visualize(X,Y)
end
```

Example 18.1 Fig. 18.1 shows the classification results of a perceptron network with $d = 2$ input nodes for a 2-D dataset and $m = 2$ output nodes that encode $2^2 = 4$ classes. The training set contains $N = 400$ data samples including 100 samples for each of the four classes labeled by $\mathbf{y} = [y_1, y_2]$. The two weight vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ are initialized randomly. During the training iteration, the weight vectors are modified if $\delta = y - \hat{y} \neq 0$, otherwise they remain the same. After nine such modifications, the four classes are completely separated by the two straight lines normal to the two weight vectors. The first panel shows the initial stage, while the subsequent panels show how the weight vectors are modified each time when $\delta \neq 0$. The darker and bigger dots represent the samples presented to the network when one of the weight vectors is modified.

Fig. 18.2 shows the classification results of a perceptron of $d = 3$ input nodes and $m = 3$ output nodes that encode $2^3 = 8$ classes. After 35 modifications the weight vectors, the perceptron is completely trained to classify all eight classes correctly. The first panel shows the initial stage, the following three panels show the weight vectors, $\mathbf{w}_1$, $\mathbf{w}_2$ and $\mathbf{w}_3$, also the normal vectors of the decision planes, from some three different viewing angles. We see that the eight classes are indeed separated by the three planes normal to the weight vectors.

The perceptron algorithm as a binary classifier is based on the assumption that the two classes are linearly separable. This limitation can be removed by the kernel method, if the algorithm is modified in such a way that all data samples

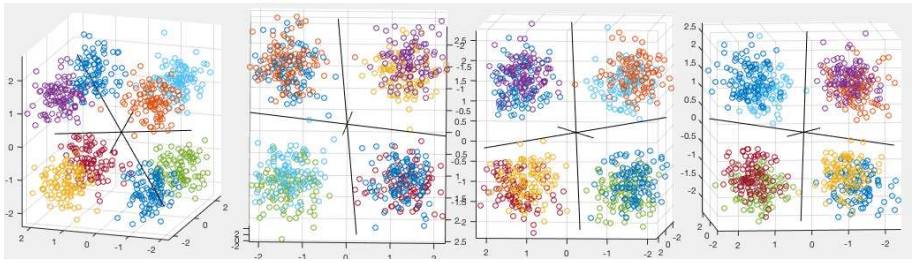


Figure 18.2 Classification by Perceptron (3-D)

appear only in the form of an inner product. Consider first the training process in which the N training samples are repeatedly presented to the network, and the weight vector is modified to become $\mathbf{w}^{new} = \mathbf{w}^{old} + y_n \mathbf{x}_n$ whenever a sample $\mathbf{x}_n$ labeled by y_n is misclassified with $\delta = y_n - \hat{y}_n \neq 0$. If the weight vector is initialized to zero $\mathbf{w} = 0$, then the weight vector by the updating rule in Eq. (18.10) can be written as a linear combination of the N training samples:

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n \tag{18.13}$$

where α_n is the number of times sample $\mathbf{x}_n$ labeled by y_n is misclassified. Upon receiving a new training sample $\mathbf{x}_l$ labeled by y_l, we have

$$f(\mathbf{x}_l) = \mathbf{w}^T \mathbf{x}_l = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n^T \mathbf{x}_l, \qquad \text{and} \qquad \hat{y}_l = \text{sign}(f(\mathbf{x}_l)) \tag{18.14}$$

and the weight vector $\mathbf{w}$ is updated by the delta-rule:

$$\begin{aligned} &\text{If} \quad \delta = y_l - \hat{y}_l \neq 0 \\ &\text{then} \quad \mathbf{w}^{new} = \mathbf{w}^{old} + y_l \, \mathbf{x}_l = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n + y_l \, \mathbf{x}_l \\ &\text{i.e.} \quad \alpha_l^{new} = \alpha_l^{old} + 1 \end{aligned} \tag{18.15}$$

We see that only $\{\alpha_1, \cdots, \alpha_N\}$ need to be updated during the training process, while the weight vector $\mathbf{w}$ in Eq. (18.13) no longer needs to be explicitly calculated. Once the training process is complete, any unlabeled $\mathbf{x}$ can be classified into either of the two classes based on $\{\alpha_1, \cdots, \alpha_N\}$:

$$\text{If} \quad \mathbf{w}^T \mathbf{x} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n^T \mathbf{x} \begin{cases} > 0 \\ < 0 \end{cases}, \quad \text{then} \quad \mathbf{x} \in \begin{cases} C_- \\ C_+ \end{cases} \tag{18.16}$$

As all data samples appear in the form of inner product in both the training and testing phase, the kernel method can be applied to replace the inner product $\mathbf{x}_n^T \mathbf{x}_m$ by a kernel function $k(\mathbf{x}_n, \mathbf{x}_m)$. Also, the discussion above for $m = 1$ output nodes can be generalized to $m > 1$ output nodes.

Here is the Matlab code segment for the most essential parts of the kernel perceptron algorithm:

```
[X Y]=Data;                 % get dataset
[d N]=size(X);              % numbers of dimensions and samples
X=[ones(1,N); X];           % augmented data
m=size(Y,1);                % number of output nodes
A=zeros(m,N);               % initial alphas for all nodes and samples
K=Kernel(X,X);              % get kernel matrix of all N samples
for it=1:nt
    n=randi([1 N]);         % random index
    x=X(:,n);               % pick a training sample
    y=Y(:,n);               % and its label
    yhat=sign((A.*Y)*K(:,n));   % get yhat
    delta=Y(:,n)-yhat;      % error = desired - actual
    for i=1:m               % for each output node
       if delta(i)~=0       % if a misclassification
           A(i,n)=A(i,n)+1; % update the corresponding alpha
       end
    end
    if ~mod(it,N)           % test for every epoch
        er=test(X,Y,A);     % percentage of misclassification
        if er<10^(-9)
            break
        end
    end
end

function er=test(X,Y,A)     % function for testing
    [d N]=size(X);          % dimension, number of samples
    m=size(Y,1);
    Ne=0;                   % number of misclassifications
    for n=1:N               % for all N training samples
        x=X(:,n);           % get the nth sample
        y=Y(:,n);           % and its label
        f=(A.*Y)*Kernel(X,x); % f(x)
        yhat=sign(f);       % yhat=sign(f(x))
        if norm(y-yhat)~=0  % misclassification at some output nodes
            Ne=Ne+1;        % increase number of misclassification
        end
    end
    er=Ne/N;                % percentage of misclassification
end
```

Example 18.2 The kernel perceptron is applied to the classification of the iris

dataset with $m = 3$ one-hot output nodes for the three classes, based on the radial basis kernel. The network is trained on half of the data and tested on the other half. The confusion matrix is shown below and the error rate is 14.67%.

$$\begin{bmatrix} 22 & 6 & 0 \\ 0 & 18 & 5 \\ 0 & 0 & 24 \end{bmatrix}$$

Example 18.3 The kernel perceptron is applied to the handwritten digit dataset with $m = 10$ one-hot output nodes, based on the radial basis kernel. The kernel perceptron is trained on half of the data, and then tested one both the training half and the testing half of the data, and the the confusion matrices are shown below, and the corresponding error rates are zero when tested on the training half of the data and 0.11 when tested on the testing half.

$$\begin{bmatrix} 116 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 108 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 108 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 116 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 113 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 122 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 103 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 108 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 109 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 116 \end{bmatrix}$$

$$\begin{bmatrix} 104 & 7 & 0 & 0 & 0 & 1 & 2 & 1 & 0 & 1 \\ 0 & 103 & 0 & 0 & 0 & 0 & 3 & 1 & 0 & 1 \\ 1 & 0 & 103 & 0 & 0 & 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & 8 & 102 & 0 & 0 & 0 & 2 & 1 & 3 \\ 1 & 1 & 0 & 0 & 103 & 2 & 0 & 0 & 4 & 2 \\ 0 & 0 & 0 & 2 & 0 & 93 & 24 & 0 & 3 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 95 & 4 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 104 & 0 & 3 \\ 1 & 0 & 1 & 1 & 3 & 2 & 1 & 3 & 93 & 4 \\ 1 & 2 & 0 & 1 & 7 & 0 & 0 & 3 & 6 & 96 \end{bmatrix}$$

The capability of the perceptron algorithm is limited in the sense that there is only one level of learning taking place between the output and input layers. This situation can be much improved by including additional learning layers between the input and output layers to form a multi-layer network, such as the back propagation which has been most widely used as a supervised learning algorithm, to be discussed in the next section.

18.2 Back Propagation

The back propagation network (BP network or BPN) is a supervised learning algorithm that finds a wide variety of applications in practice. In the most general sense, a BPN can be used as an associator that learns the relationship between two sets of patterns represented in vector forms. Specifically in classification, similar to the perceptron network, the BPN as a classifier is based on the training set $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, of which each pattern $\mathbf{x}_n$ is associated with the corresponding pattern, an m-dimensional vector $\mathbf{y}_n$ in $\mathbf{Y} = [\mathbf{y}_1, \cdots, \mathbf{y}_N]$, as its class identity labeling, indicating to to which of the K classes $\{C_1, \cdots, C_K\}$ pattern $\mathbf{x}_i$ belongs.

Different from the perceptron in which there is only one level of learning taking place between the output and input layers in terms of the weights, a BPN is a multi-layer (three or more) hierarchical structure composed of the input layer, one or more hidden layers, and the output layer, in which learning takes place in multiple levels between two consecutive layers. Consequently, a BPN can therefore be more flexible and powerful than the two-layer perceptron network, and more capable to learn more complicated and patterns.

In the following, before we consider the general BPN containing multiple hidden layers, we will first derive the back propagation algorithm for the simplest BPN with only one hidden layer in between the input and output layers, as shown in Fig. 18.3. We note that there are $d+1$ input nodes for input $\mathbf{x} = [x_1, \cdots, x_d]^T$ and a constant $x_0 = 1$, and $l + 1$ hidden nodes that generate $\mathbf{z} = [z_1, \cdots, z_l]^T$ and a constant $z_0 = 1$, while the output layer simply contain m nodes for the output $\mathbf{y} = [y_1, \cdots, y_m]^T$.

Each node in the hidden (excluding $z_0 = 1$) and output layers is fully connected to all nodes in the previous layer. When one of the N training patterns $\mathbf{x}_n$ is presented to the input layer of the BPN, an m-D vector $\hat{\mathbf{y}}_n = f(\mathbf{x}_n, \mathbf{W}^h, \mathbf{W}^o)$ is produced at the output layer as the corresponding response to the input $\mathbf{x}_n$. Here $\mathbf{W}^h = [\mathbf{w}_1^h, \cdots, \mathbf{w}_l^h]$ (a $(d + 1) \times l$ matrix) and $\mathbf{W}^o = [\mathbf{w}_1^o, \cdots, \mathbf{w}_m^o]$ (a $(l + 1) \times m$ matrix) are the model parameters containing the augmented weight vectors for both the hidden and output layers, to be determined in the training process based on the training set $\mathbf{X}$ and corresponding labeling $\mathbf{Y}$, so that its output $\hat{\mathbf{y}}_n$ as a function of the current input $\mathbf{x}_n$ matches the desired output or the target, the labeling $\mathbf{y}_n$ as closely as possible. Once the BPN is fully trained, any unlabeled pattern $\mathbf{x}$ can be classified into one of the K classes corresponding to the minimum $\delta = ||\mathbf{y} - \hat{\mathbf{y}}||$. Note that different from the perceptron network, the output of the m output nodes take real values in general, although they may still be binary (such as one-hot or binary encoding for class labeling) if so desire.

Specifically, the training of the BPN is an iteration of the following two-phase process:

- **The feedforward pass:**

  A randomly selected sample $\mathbf{x}_n$ labeled by $\mathbf{y}_n$ is presented to the input

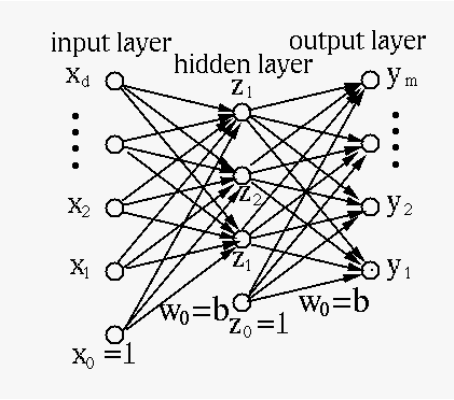


Figure 18.3 Back Propagation Network

layer and forwarded through the weighted connections to the hidden layer and then the output layer to produce output $\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}^h, \mathbf{W}^o)$.

- **The backward error backpropagation:**

  The squared error $\varepsilon = ||\mathbf{y}_n - \hat{\mathbf{y}}_n||^2/2$ measuring the difference between the actual output $\hat{\mathbf{y}}_n$ and the desired output $\mathbf{y}_n$ as the target is propagated backward from the output layer through the hidden layer to the input layer, during which the weights of both the output and hidden layers are modified so that ε will be reduced when the same or similar pattern is presented to the input layer in the future.

This two-phase process is iteratively carried out until eventually the error is minimized for all training samples and BPN is properly trained.

We now consider the specific computation taking place in both the forward and backward passes.

- The forward pass from input $\mathbf{x} = [x_1, \cdots, x_d]^T$ represented to the input layer to the output $\hat{\mathbf{y}}_n = [y_1, \cdots, y_m]^T$ produced by the output layer:
  - From input layer to hidden layer:

$$z_j = g(a_j^h) = g\left(\sum_{k=1}^{d} w_{jk}^h x_k + b_j^h\right) = g\left(\sum_{k=0}^{d} w_{jk}^h x_k\right) = g(\mathbf{x}^T \mathbf{w}_j^h) \qquad (j = 1, \cdots, l) \tag{18.17}$$

  where both $\mathbf{x} = [x_0 = 1, x_1, \cdots, x_d]^T$ and $\mathbf{w}_j^h = [w_{j0}^h = b_j^h, w_{j1}^h, \cdots, w_{jd}^h]^T$ are augmented, and $a_j^h = \mathbf{x}^T \mathbf{w}_j^h$ is the activation of the jth hidden layer node. These l equations can be expressed in vector form:

$$\mathbf{z} = \mathbf{g}\left(\mathbf{W}^h \mathbf{x}\right) \tag{18.18}$$

  where $\mathbf{z} = [z_1, \cdots, z_l]^T$, and $\mathbf{W}^h = [\mathbf{w}_1^h, \cdots, \mathbf{w}_l^h]^T$ is an $l \times (d+1)$ matrix.

– From hidden layer to output layer:

$$\hat{y}_i = g(a_i^o) = g\left(\sum_{j=1}^{l} w_{ij}^o z_j + b_i^o\right) = g\left(\sum_{j=0}^{l} w_{ij}^o z_j\right)$$
$$= g\left(\sum_{j=0}^{l} w_{ij}^o g\left(\sum_{k=0}^{d} w_{jk}^h x_k\right)\right) \qquad (i = 1, \cdots, m) \quad (18.19)$$

which can also be written in vector form as:

$$\hat{y}_i = g(\mathbf{z}^T \mathbf{w}_i^o) \quad (18.20)$$

where both $\mathbf{z} = [z_0 = 1, z_1, \cdots, z_l]^T$ and $\mathbf{w}_i^o = [w_{i0}^o = b_i^o, w_{i1}^o, \cdots, w_{il}^o]^T$ are augmented, and $a_i^o = \mathbf{z}^T \mathbf{w}_i^o$ is the activation of the ith output layer node. These m equations can be expressed in vector form:

$$\hat{\mathbf{y}} = \mathbf{g}\left(\mathbf{W}^o \mathbf{z}\right) \quad (18.21)$$

where $\hat{\mathbf{y}} = [\hat{y}_1, \cdots, y_m]^T$, and $\mathbf{W}^o = [\mathbf{w}_1^o, \cdots, \mathbf{w}_m^o]^T$ is an $m \times (l+1)$ matrix.

- The backward error propagation from error ε to the input layer:

The squared error $\varepsilon = ||\mathbf{y} - \hat{\mathbf{y}}||^2/2$ can be written as a function of the output layer weights w_{ij}^o $(i = 1, \cdots, m,\ j = 0, 1, \cdots, l)$ and hidden layer weights w_{jk}^h $(j = 1, \cdots, l,\ k = 0, 1, \cdots, d)$:

$$\varepsilon = \frac{1}{2}||\mathbf{y} - \hat{\mathbf{y}}||^2 = \frac{1}{2}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{m}[g(a_i^o) - y_i]^2$$
$$= \frac{1}{2}\sum_{i=1}^{m}\left[g\left(\sum_{j=0}^{l} w_{ij}^o z_j\right) - y_i\right]^2 = \frac{1}{2}\sum_{i=1}^{m}\left[g\left(\sum_{j=0}^{l} w_{ij}^o\, g(a_j^h)\right) - y_i\right]^2$$
$$= \frac{1}{2}\sum_{i=1}^{m}\left[g\left(\sum_{j=0}^{l} w_{ij}^o\, g\left(\sum_{k=0}^{d} w_{jk}^h x_k\right)\right) - y_i\right]^2 \quad (18.22)$$

Similar to the objective function of the ridge regression considered in Section 4.2, an additional regularization term weighted by the *weight decay parameter* λ can be included in the objective function to penalize large weights and thereby prevent overfitting:

$$J(\mathbf{W}^h, \mathbf{W}^o) = \varepsilon + \frac{\lambda}{2}\left[||\mathbf{W}^h||_2^2 + ||\mathbf{W}^o||_2^2\right] \quad (18.23)$$

where the two weight matrices in the regularization term should exclude the first column for the bias term b which need not to be forced to be small, and $||\mathbf{W}||_2 = \sqrt{\sum_i \sum_j w_{ij}^2}$ is the *Frobenius norm* (Section A.4.2).

The gradient descent method is used to modify first the output layer weights and then the hidden layer weights to iteratively reduce the objective function J in the following steps:

– Find the gradient of J with respect to the output layer weights w_{ij}^o $(i = 1, \cdots, m,\ j = 0, 1, \cdots, l)$ by the chain rule:

$$\begin{aligned}\frac{\partial J}{\partial w_{ij}^o} &= \frac{\partial \varepsilon}{\partial \hat{y}_i}\,\frac{\partial \hat{y}_i}{\partial a_i^o}\,\frac{\partial a_i^o}{\partial w_{ij}^o} + \lambda w_{ij}^o = (\hat{y}_i - y_i)\; g'(a_i^o)\; z_j + \lambda w_{ij}^o \\ &= -\delta_i\; g'(a_i^o)\; z_j + \lambda w_{ij}^o \end{aligned} \tag{18.24}$$

where $\delta_i = y_i - \hat{y}_i$.

– Update w_{ij}^o $(i = 1, \cdots, m,\ j = 0, 1, \cdots, l)$ to reduce J by gradient descent method with learning rate or step size η:

$$\begin{aligned} w_{ij}^o &\Leftarrow w_{ij}^o - \eta \frac{\partial J}{\partial w_{ij}^o} = w_{ij}^o - \eta \left(-\delta_i\; g'(a_i^o)\; z_j + \lambda w_{ij}^o\right) \\ &= w_{ij}^o - \eta \left(-d_i^o\; z_j + \lambda w_{ij}^o\right) \end{aligned} \tag{18.25}$$

where $d_i^o = \delta_i g'(a_i^o)$, or in matrix form:

$$\mathbf{W}^o \Leftarrow \mathbf{W}^o - \eta \left(-\mathbf{d}^o \mathbf{z}^T + \lambda \mathbf{W}^o\right) = \mathbf{W}^o + \eta \mathbf{d}^o \mathbf{z}^T - \eta \lambda \mathbf{W}^o \tag{18.26}$$

where $\mathbf{d}^o = [d_1^o, \cdots, d_m^o]^T$ is the elementwise (Hadamard) product $\delta \odot \mathbf{g}'(\mathbf{a}^o)$, and $\mathbf{d}^o \mathbf{z}^T = \mathbf{d}^o \otimes \mathbf{z}$ is the outer product of $\mathbf{d}^o$ and $\mathbf{z} = [z_0 = 1, z_1, \cdots, z_l]$.

– Find the gradient of J with respect to the hidden layer weights w_{jk}^h $(j = 1, \cdots, l,\ k = 0, 1, \cdots, d)$ by the chain rule:

$$\begin{aligned}\frac{\partial J}{\partial w_{jk}^h} &= \frac{\partial \varepsilon}{\partial \hat{y}_i}\,\frac{\partial \hat{y}_i}{\partial a_i^o}\,\frac{\partial a_i^o}{\partial z_j}\,\frac{\partial z_j}{\partial a_j^h}\,\frac{\partial a_j^h}{\partial w_{jk}^h} + \lambda w_{jk}^h = \sum_{i=1}^{m} (\hat{y}_i - y_i)\,\frac{\partial \hat{y}_i}{\partial a_i^o}\,\frac{\partial a_i^o}{\partial z_j}\,\frac{\partial z_j}{\partial a_j^h}\,\frac{\partial a_j^h}{\partial w_{jk}^h} + \lambda w_j^h \\ &= -\sum_{i=1}^{m} \delta_i g'(a_i^o) w_{ij}^o\; g'(a_j^h) x_k + \lambda w_{jk}^h = -\left(\sum_{i=1}^{m} d_i^o w_{ij}^o\right) g'(a_j^h) x_k + \lambda w_{jk}^h \\ &= -\delta_j^h\; g'(a_j^h) x_k + \lambda w_{jk}^h = -d_j^h x_k + \lambda w_{jk}^h \end{aligned} \tag{18.27}$$

where we have defined

$$\delta_j^h = \sum_{i=1}^{m} d_i^o w_{ij}^o, \qquad d_j^h = \delta_j^h g'(a_j^h) = \left(\sum_{i=1}^{m} d_i^o w_{ij}^o\right) g'(a_j^h), \qquad (j = 1, \cdots, l) \tag{18.28}$$

or in matrix form:

$$\begin{bmatrix} \delta_1^h \\ \vdots \\ \delta_l^h \end{bmatrix} = \begin{bmatrix} w_{11}^o & \cdots & w_{1m}^o \\ \vdots & \ddots & \vdots \\ w_{l1}^o & \cdots & w_{lm}^o \end{bmatrix} \begin{bmatrix} d_1 \\ \vdots \\ d_m \end{bmatrix} = \mathbf{W}^o \mathbf{d}^o \tag{18.29}$$

Here $\mathbf{W}^o$ is an $m \times l$ matrix, the same as that defined above but with the first column of b_j's removed.

– Update w_{jk}^h $(j = 1, \cdots, l,\ k = 0, 1, \cdots, d)$ to reduce J by gradient descent method:

$$w_{jk}^h \Leftarrow w_{jk}^h - \eta \frac{\partial J}{\partial w_{jk}^h} = w_{jk}^h - \eta \left(-\delta_j^h\ g'(a_j^h) x_k + \lambda w_{jk}^h\right)$$
$$= w_{jk}^h - \eta \left(-d_j^h x_k + \lambda w_{jk}^h\right) \tag{18.30}$$

or in matrix form:

$$\mathbf{W}^h \Leftarrow \mathbf{W}^h - \eta \left(-\mathbf{W}^o \mathbf{d}^o \odot \mathbf{g}'(\mathbf{a}^h)\mathbf{x}^T + \lambda \mathbf{W}^h\right) = \mathbf{W}^h + \eta \mathbf{d}^h \mathbf{x}^T - \eta\lambda \mathbf{W}^h \tag{18.31}$$

where $\mathbf{d}^h = \mathbf{W}^o \mathbf{d}^o \odot \mathbf{g}'(\mathbf{a}^h)$ is the elementwise product of vector $\mathbf{W}^o \mathbf{d}^o$ and $\mathbf{g}'(\mathbf{a}^h)$, and $\mathbf{d}^h \mathbf{x}^T = \mathbf{d}^h \otimes \mathbf{x}$ is the outer product of vectors $\mathbf{d}^h$ and $\mathbf{x}$.

In summary, here are the steps in each iteration:

1. Input a randomly selected pattern $[x_1, \cdots, x_d]^T$, construct $d+1$ dimensional vector $\mathbf{x} = [1, x_1, \cdots, x_d]^T$;
2. Compute $\mathbf{z} = \mathbf{g}(\mathbf{W}^h \mathbf{x})$, and construct $l+1$ dimensional vector $\mathbf{z} \leftarrow [1, \mathbf{z}]$;
3. Compute $\hat{\mathbf{y}} = \mathbf{g}(\mathbf{W}^o \mathbf{z})$;
4. Get elementwise product $\mathbf{d}^o = (\mathbf{y} - \hat{\mathbf{y}}) \odot \mathbf{g}'(\mathbf{a}^o) = \delta \odot \mathbf{g}'(\mathbf{a}^o)$;
5. Update output weights $\mathbf{W}^o \Leftarrow \mathbf{W}^o + \eta\ \mathbf{d}^o \mathbf{z}^T - \eta\lambda \mathbf{W}^o$;
6. Get elementwise product $\mathbf{d}^h = \mathbf{W}_{m\times l}^o \mathbf{d}^o \odot \mathbf{g}'(\mathbf{a}^h) = \delta^h \odot \mathbf{g}'(\mathbf{a}^h)$, where $\mathbf{W}_{m\times l}^o$ is the same as $\mathbf{W}^o$ but with its first column removed.
7. Update hidden weights $\mathbf{W}^h \Leftarrow \mathbf{W}^h + \eta\ \mathbf{d}^h \mathbf{x}^T - \eta\lambda \mathbf{W}^h$;
8. Terminate the iteration if the error ε is acceptably small for all of the training patterns. Otherwise repeat the above with another pattern in the training set.

The Matlab code for the essential part of the BPN algorithm is listed below. Array X contains C classes each with K samples, array Y are the labelings of the $C * K$ training samples, array W contains the $N+1$ dimensional weight vectors for the M output nodes. Also, L is the number of hidden nodes, *eta* is the learning rate between 0 and 1, and *tol* is the tolerance for the termination of the learning iteration (e.g., 0.01).

```
function [Wh, Wo, g]=backPropagate(X,Y)
    syms x;
    g=1/(1+exp(-x));              % Sigmoid activation function
    dg=diff(g);                   % its derivative function
    g=matlabFunction(g);
    dg=matlabFunction(dg);
    [d,N]=size(X);                % numbers dimensions and samples
    X=[ones(1,N); X];             % augment X by adding a row of x0=1
    Wh=1-2*rand(L,d+1);           % Initialize hidden layer weights
    Wo=1-2*rand(M,L+1);           % Initialize output layer weights
```

```
    er=inf;
    while er > tol
        I=randperm(N);           % random permutation of samples
        er=0;
        for n=1:N                % for all N samples for an epoch
            x=X(:,I(n));         % pick a training sample
            ah=Wh*x;             % activation of hidden layer
            z=[1; g(ah)];        % augment z by adding z0=1
            ao=Wo*z;             % activation to output layer
            yhat=g(ao);          % output of output layer
            delta=Y(:,I(n))-yhat % delta error
            er=er+norm(delta)/N  % test error
            do=delta.*dg(ao);
            Wo=Wo+eta*do*z';     % update weights for output layer
            dh=(Wo(:,2:L+1)'*do).*dg(ah);   % delta of hidden layer
            Wh=Wh+eta*dh*x';     % update weights for hidden layer
        end
    end
end
```

The training process of BP network can also be considered as a data modeling problem as in regression to fit the given data $\{(\mathbf{x}_i, \mathbf{y}_i),\ (i = 1, \cdots, N)\}$ by a function with the weights of both the hidden and output layers as the parameters:

$$\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{W}^h, \mathbf{W}^o) \tag{18.32}$$

The goal is to find the optimal parameters $\mathbf{W}^h$ and $\mathbf{W}^o$ that minimize the difference $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}}$ between the desired and the actual outputs. The Levenberg-Marquardt algorithm discussed previously can be used to obtain the parameters (e.g., the Matlab function *trainlm* `http://www.mathworks.com/help/nnet/ref/trainlm.html`).

The three-layer BPN containing a single hidden layer discussed above can be easily generalized to a multilayer BPN containing any number of hidden layers (e.g., for a deep learning network) by simply repeating steps 6 and 7 for the single hidden layer in the algorithm list above. We assume in addition to the input layer, there are in total L learning layers including all the hidden layers and the output layer, indexed by $l = 1, \cdots, L$. Then step 6 above becomes:

$$\mathbf{d}^{(l)} = \delta^{(l)} \cdot \mathbf{g}'(\mathbf{a}^{(l)}) \qquad (l = 1, \cdots, L) \tag{18.33}$$

where $\mathbf{a}^{(l)} = (\mathbf{W}^{(l)}\mathbf{z}^{(l)})$ is the activation of all nodes in the lth layer, $\mathbf{z}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{z}^{(l-1)}$ is the input to the lth layer, output from the (l-1)th layer ($\mathbf{z}^{(1)} = \mathbf{x}$), and

$$\delta^{(l)} = \begin{cases} \mathbf{y} - \hat{\mathbf{y}} & (l = L) \\ \mathbf{W}^{(l)}\mathbf{d}^{(l-1)} & (1 \le l < L) \end{cases} \tag{18.34}$$

Then the same step 7 for updating the weights can be carried out based on the

gradient vector:

$$\mathbf{d}^{(l)} \otimes \mathbf{z}^{(l)} = \mathbf{d}^{(l)}(\mathbf{z}^{(l)})^T \tag{18.35}$$

The Matlab code segment below is for the BP network of multiple hidden layers, in which L is the total number of learning layers and $m(l)$ is the number of nodes in the lth layer ($l = 1, \cdots, L$):

```
W={1-2*rand(m(1),d+1)};           % initial weights for first layer
for l=2:L
    W{l}=1-2*rand(m(l),m(l-1)+1); % initial weights for other layers
end
er=inf;
d={};
while er > tol
    I=randperm(N);                % random permutation of samples
    er=0;
    for n=1:N                     % N samples for an epoch
        z={1;X(:,I(n))};          % pick a training sample
        a={W{1}*z{1}};            % activation of first layer
        for l=2:L                 % the forward pass
            z{l}=[1;g(a{l-1})];   % input to the lth layer
            a{l}=W{l}*z{l};       % activation of the lth layer
        end
        yhat=g(a{L});             % actual output of last layer
        delta=Y(:,I(n))-yhat;     % delta error
        er=er+norm(delta)/N;      % test error
        d{L}=delta.*dg(a{L});     % d for output layer
        W{L}=W{L}+eta*d{L}*z{L}'; % updddate weights for output layer
        for l=L-1:-1:1            % the backward pass
            d{l}=(W{l+1}(:,2:end)'*d{l+1}).*dg(a{l}); % d of hidden layers
            W{l}=W{l}+eta*d{l}*z{l}'; % updddate weights for hidden layers
        end
    end
end
```

Example 18.4 Figs. 18.4 and 18.5 show the classification results of two 2-D datasets with 2 and 3 classes previously used in Examples 13.6 and 13.7 for the naive Bayes classifier (with quadratic decision boundaries). The error rates are respectively 13% and 11.5%, and the confusion matrices are shown below. Iinterestingly, in the case of the 3-class dataset, there is a green region sandwiched between the blue and red ones in the results generated by both the naive Bayes and back propagation methods based on very different learning algorithms.

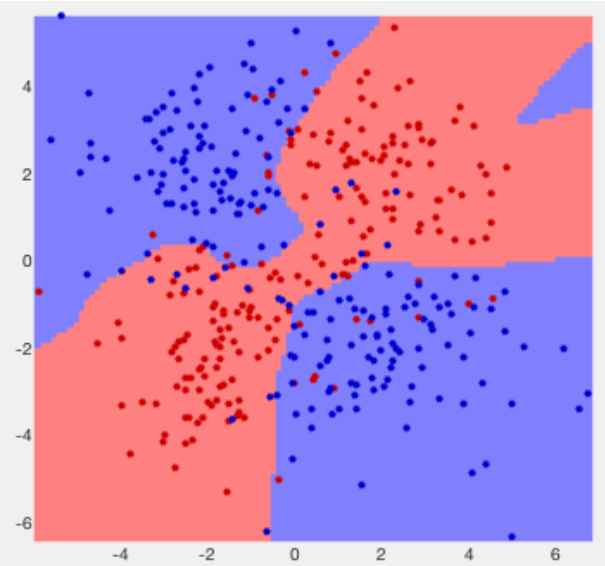


Figure 18.4 BP Network Classification of the XOR Dataset

$$\begin{bmatrix} 176 & 24 \\ 22 & 178 \end{bmatrix} \qquad \begin{bmatrix} 185 & 14 & 1 \\ 5 & 181 & 14 \\ 11 & 33 & 156 \end{bmatrix}$$

Example 18.5 The back propagation network is applied to the classification of the handwritten digit dataset. Half of the $N = 2240$ samples in the dataset is used for training while the other half used for testing. The classification results are shown by the two confusion matrices below, when the network is tested on half of the samples used for training (left) and on the remaining half reserved for testing (right). Note that the error rate $74/1120 = 6.6\%$ in the latter case is understandably higher than the error $27/1120 = 2.4\%$ for the former, as the network may overfit half of the samples used for training and does not generate too well to the other half of the samples it has never seen.

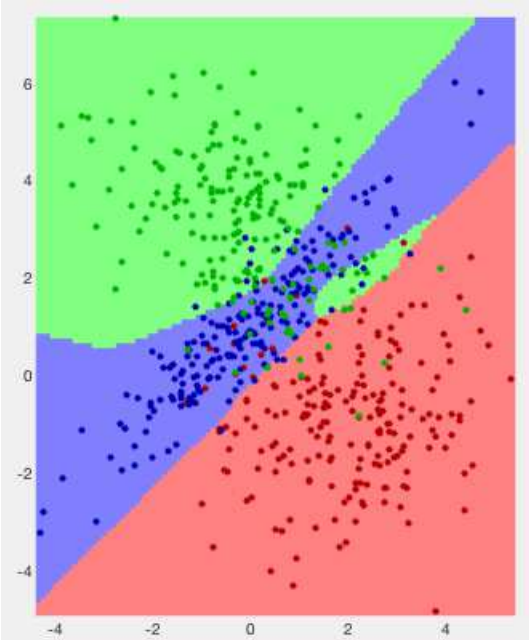


Figure 18.5 BP Network Classification of a Three-Class Dataset

$$\begin{bmatrix} 113 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 107 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 106 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 113 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 113 & 8 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 120 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 102 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 1 & 0 & 103 & 0 & 1 \\ 0 & 1 & 0 & 1 & 2 & 0 & 0 & 0 & 104 & 1 \\ 1 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 112 \end{bmatrix}$$

$$\begin{bmatrix} 100 & 1 & 0 & 0 & 0 & 4 & 3 & 0 & 0 & 0 \\ 0 & 111 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 113 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 2 & 1 & 1 & 100 & 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 0 & 105 & 0 & 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 0 & 3 & 95 & 0 & 0 & 2 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 & 118 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 105 & 1 & 5 \\ 1 & 1 & 2 & 1 & 1 & 5 & 0 & 2 & 97 & 5 \\ 0 & 2 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 102 \end{bmatrix}$$

Before leaving this section, we further make the following comments regarding how the back propagation network, and in general, multilayer feed forward network, is closely related to the general method of regression analysis discussed before.

- The computational model for a single neuron in Eq. (17.3) based on the logistic activation function in Eq. (17.5) is actually the same as Eq. (7.6) in logistic regression in Section 7.1:

$$g\left(\sum_{j=1}^{d} w_j x_j + b\right) = g(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}} \tag{18.36}$$

  treated as the conditional probability $p(\hat{y}=1|\mathbf{x},\mathbf{w})$. Both methods are to determine the model parameter $\mathbf{w}$, but based on different methods.
- A supervised learning neural network can be considered as a function $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x},\mathbf{w})$ as a model of the training dataset $\mathbf{D} = \{\mathbf{X},\mathbf{y}\}$, while the parameter $\mathbf{w}$ can be found by solving an optimization problem (Chapter 5), by either the least squares method that minimizes $\varepsilon = ||\mathbf{y}-\hat{\mathbf{y}}||^2$, or the maximum likelihood method that maximizes $L(\mathbf{w}) = P(\mathbf{y}|\mathbf{X},\mathbf{w})$, equivalent to minimizing $-l(\mathbf{w}) = -\log L(\mathbf{w})$, which can be interpreted as the cross entropy between the model distribution based on $\mathcal{D}$ and the unknown ground true distribution, as discussed in Section 4.1.
- As a neural network can be considered as a model for the function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ relating $\mathbf{y}$ to $\mathbf{x}$ in the given dataset $\mathcal{D}$, a fundamental question to ask is whether such a network is able to model any function. This question is answered by the *universal approximation theorem*, stating that a feedforward network with arbitrary width (number of nodes in a hidden layer) or depth (number of hidden layers) can represent any function with arbitrary accuracy.

  While this theorem is formally proven, it can also be understood intuitively. Recall the method of linear regression based on basis functions in Chapter 5, by which a function is approximated by a repression function as a linear combination of K basis functions in Eq. (5.47):

$$\hat{y}_m = \sum_{k=1}^{K} w_{mk}^o\, \varphi_k(\mathbf{x}) = \sum_{k=1}^{K} w_{mk}^o\, g\left(\sum_{i=0}^{d} w_{ki}^h x_i\right) \tag{18.37}$$

  which is in the same form as in Eq. (18.19) for the three-layer feed forward network discussed above, with a linear activation function for the output layer. We see the the output of the hidden layer nodes are the basis functions, and the output of an output layer node is a linear combination of such basis functions. As shown in Example 5.10 there, a given function can be approximated with progressively higher accuracy as more basis functions, i.e., more hidden layer nodes, are used, if the width of the network is unbounded.

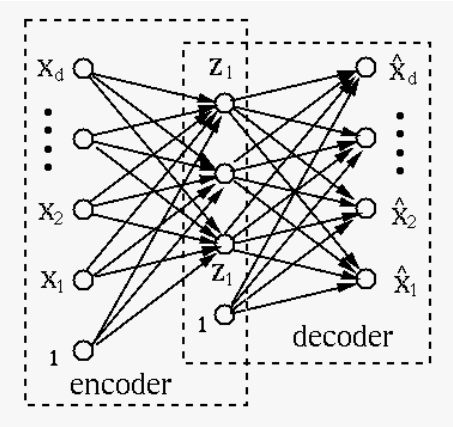


Figure 18.6 Thee layer Network for Autoenoder

If the width of the network is bounded but its depth is not, we can alternatively increase the number of layers of the network so that in the equation above we can have as many levels of summations as needed, i.e., the total number of terms in these summations can increased without a bound, then by the same argument, such a network can again approximate a given function with arbitrary accuracy. But as a word of warning, while trying to match the given dataset for better accuracy, watch out for the overfitting problem.

18.3 Autoencoder

Autoencoder is a neural network method that learns to represent the patterns in a given dataset in such a way that both the dimensionality and the noise can be reduced, and it can therefore be considered as a method for feature selection and dimensionality reduction, similar to the KLT method.

An autoencoder as shown in Fig. 18.6 is similar to a back propagation network in that it iteratively updates the weights in all hidden and output layers by back propagation so that eventually its outputs are as desired. But different from a back propagation network that learns to associate a pattern $\mathbf{x}_n$ to another pattern $\mathbf{y}_n$ as its labeling, the autoencoder is an unsupervised learning method based on unlabeled data, and it learns to associate each pattern $\mathbf{x}_n$ to itself.

The purpose of the autoencoder is not to simply copy the data. Instead, we desire to reduce the number of hidden layer nodes as much as possible, by imposing constraints such as sparsity to the hidden layer(s) in terms of the number of active nodes, so that there are many fewer nodes in the hidden layers when compared to the input and output layers. By doing so, the hidden layers becomes the *bottleneck* of the multi-layer network, which can then be considered

as an *encoder* by which the dimensionality of the data presented to the input layer to be reproduced by the output layer is converted into a code in a much reduced dimensional space in the hidden layer. By this approach, the data can be much compressed while the information contained in it is mostly preserved, and certain potential structures inherent in the dataset may be revealed, very much like dimensionality reduction achieved PCA based on KLT.

The computation taking place in an autoencoder can therefore be considered as an encoding/decoding process, with encoding taking place between the input and hidden layers, and decoding taking place between the hidden and output layers. Due to the reduced number of nodes in the hidden layers, the data size can be significantly reduced for data compression, if some of the components of data samples $\mathbf{x}$ in the dataset are correlated, i.e., there is some redundancy in the data.

While the methods of autoencoder and PCA based on KLT are similar to each other in that both can compress the data without losing much information, they are different in that the PCA is a linear method, as the principal components are linear combinations of the data components, while the signals represented by the hidden layer of the autoencoder are nonlinear functions of the data components (due to the nonlinear activation function). Consequently the autoencoder can be a more powerful method than the PCA, but with the cost of higher computational complexity.

We now consider specifically how an autoencoder works. Same as in back propagation, a data vector $\mathbf{x}$ presented to the input layer is forward propagated first to the hidden layer (same as Eq. (18.18)):

$$\mathbf{z} = \mathbf{g}(\mathbf{W}^h \mathbf{x}), \qquad \text{or} \qquad z_j = g\left(\sum_{k=0}^{d} w_{jk}^h x_k\right) \qquad (j = 1, \cdots, l) \tag{18.38}$$

and then further to the output layer (same as Eq. (18.21)):

$$\hat{\mathbf{x}} = \mathbf{g}(\mathbf{W}^o \mathbf{z}) \qquad \text{or} \qquad \hat{x}_i = g\left(\sum_{j=0}^{l} w_{ij}^o z_k\right) \qquad (i = 1, \cdots, m) \tag{18.39}$$

Also same as in back propagation, the weights in both $\mathbf{W}^h$ and $\mathbf{W}^o$ are updated during training in the back propagation process so that the following objective function for the squared error between the input and desired output, the same as the input, is minimized:

$$\varepsilon = \frac{1}{2}||\mathbf{x} - \hat{\mathbf{x}}||^2 \tag{18.40}$$

Here we assume there is only a single hidden layer for convenience, but the computation for both the forward and backward passes can be easily extended to multiple layers.

By limiting the number l of the hidden layer to be smaller than that of input and output layer, the dimensionality of the data can be reduced from d for the

dimensionality of the input $\mathbf{x} = [x_1, \cdots, x_d]^T$ to l, the number of hidden layer nodes, while most of the information in the original data is reserved.

It is interesting to compare the autoencoder and the PCA based on KLT transform in terms of how they map a given data vector $\mathbf{x}$ in the original d-dimensional space to a lower-dimensional space. Recall the KLT previously considered:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_d \end{bmatrix} = \mathbf{V}^T\mathbf{x} = \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_d^T \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{v}_1^T\mathbf{x} \\ \vdots \\ \mathbf{v}_d^T\mathbf{x} \end{bmatrix} \tag{18.41}$$

In KLT, the ith component $y_i = \mathbf{v}_i^T\mathbf{x}$ of $\mathbf{y}$ is the inner product of the ith eigenvector $\mathbf{v}_i$ and the input $\mathbf{x}$, the projection of $\mathbf{x}$ onto $\mathbf{v}_i$ if it is normalized. On the other hand, in a neural network, the inner product of the weight vector $\mathbf{w}_i$ and the input $\mathbf{x}$ is further mapped to $z_i = g(\mathbf{w}_i^T\mathbf{x})$. If, specially, the activation function $g(u) = u$ is linear (an identical function), the weight vectors $\mathbf{w}_i$ play the same role as the eigenvectors $\mathbf{v}_i$ as the basis vectors that span the space, i.e., autoencoder and PCA are essentially the same. But as in general, the activation function $g(u)$ is nonlinear, the autoencoder is a nonlinear mapping, of which the linear KLT (a rotation in the space) is a special case.

Dimension reduction exemplified by the KLT is effective if the components of the pattern vectors are highly correlated, i.e., some of them may be redundant as they carry similar information. However, if all components carry their own independent information, dimensionality reduction will inevitably cause significant information loss. Similarly in an autoencoder, if the components of the data points are mostly independent of each other with little redundancy, they may not be reproduced as the output of the autoencoder if the number of hidden layer nodes is too small.

When the data components are correlated, it is possible to achieve the desired low sparsity of the hidden layer of the autoencoder by using only a small number of hidden nodes, or allowing only a small fraction of them to be active (over different input patterns). The second approach can be realized by (a) keeping only a small number of hidden nodes with maximum outputs while setting the rest to zero, or (b) forcing the average activation of the hidden nodes to be low. This can be achieved by modify the regularization term in Eq. (18.23) for small weights so that it acts to force the average activation level low, as either of the following:

- Minimize the following L1 or L2 average of all activations:

$$J = \varepsilon + \lambda \sum_{j=1}^{l} |a_j|, \qquad J = \varepsilon + \frac{\lambda}{2} \sum_{j=1}^{l} a_j^2 \tag{18.42}$$

- Minimize the difference between the actual and desired average activation:

$$J = \varepsilon + \sum_{j=1}^{l} KL(\rho||\hat{\rho}_j)$$
$$= \varepsilon + \sum_{j=1}^{l} \left[\rho \log\left(\frac{\rho}{\hat{\rho}_j}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_j}\right)\right] \tag{18.43}$$

Here ρ is the desired average activation of the hidden layer, a small value close to zero (e.g., $\rho = 0.05$), and $\hat{\rho}_j$ is the actual average activation of the jth hidden node over all N samples in the given dataset:

$$\hat{\rho}_j = \frac{1}{N}\sum_{n=1}^{N} \rho_j(\mathbf{x}_n) = \frac{1}{N}\sum_{n=1}^{N} g(a_j) = \frac{1}{N}\sum_{n=1}^{N} g\left(\sum_{k=0}^{d} w_{jk}^h x_k\right) \qquad (j = 1, \cdots, l) \tag{18.44}$$

where $\rho_j(\mathbf{x}_n) = g(a_j) = g\left(\sum_k w_{jk}^h x_k\right)$ is the activation level of the node while responding to the nth input sample $\mathbf{x}_n$. Also, ρ is treated as the probability for some binary random variable (with Bernoulli distribution) to be 1, and $1-\rho$ the probability for it to be 0, and so is $\hat{\rho}_j$ for another binary variable. We therefore see that the regularization term in Eq. (18.43) is the sum of the KL-divergence (Section B.2.2) between the two distributions for all hidden nodes, measuring the overall difference between the desired and actual sparsities in terms of ρ and $\hat{\rho}$. By minimizing the objective function including such a regularization term, the sparsity of the hidden layer in terms of the average activation is forced to be as low as desired.

The plot in Fig. 18.7 compares the KL-divergence and the quadratic function $a(\rho - \hat{\rho})^2$. While both functions are minimized to zero when $\hat{\rho} = \rho = 0.3$, they behave differently in the range of $[0,\ 1]$ in that the KL-divergence becomes significantly larger than the quadratic function when $\hat{\rho}$ are very different from $\rho = 0.3$.

Now the objective function for the autoencoder can be written as the sum of the error ε and the regularization term above:

$$J(\mathbf{W}^h, \mathbf{W}^o) = \varepsilon + \lambda \sum_{j=1}^{l} KL(\rho||\hat{\rho}_j) \tag{18.45}$$

During the iterative learning the weights are updated based on gradient descent method, same as in back propagation. Given a specific input $\mathbf{x}$ we first get the output $\hat{\mathbf{x}}$ through the forward pass, and then modify the weights of both the output and hidden layers in the backward pass based on the gradient of the objective function $\varepsilon = ||\mathbf{x} - \hat{\mathbf{x}}||^2/2$ with respect to the weights. While the computation for updating the weights for the output layer is the same as steps 4 and 5 in the algorithm of back propagation listed in Section 18.2, the computation for the hidden layer weights as shown below is different from steps 6 and 7 for back propagation due to the regularization term:

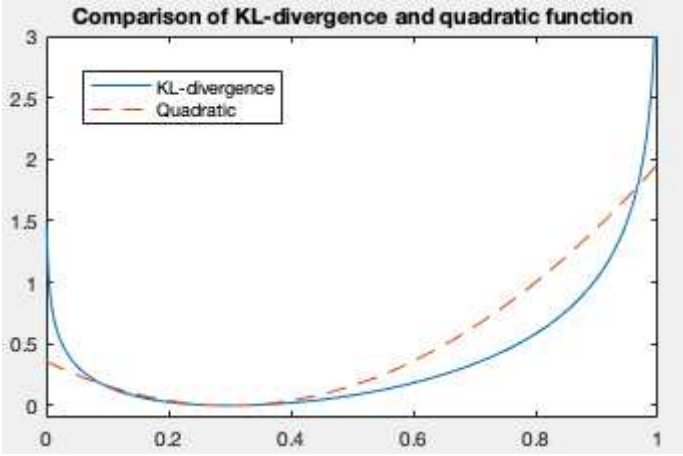


Figure 18.7 Comparison of KL-divergence and Quadratic Function

$$\frac{\partial}{\partial w_{jk}^h} J(\mathbf{W}^h, \mathbf{W}^o) = \frac{\partial \varepsilon}{\partial w_{jk}^h} + \lambda \frac{\partial}{\partial w_{jk}^h} \left[\sum_{j=1}^{l} KL(\rho || \hat{\rho}_j) \right] \tag{18.46}$$

The first term above is the same as as that in Eq. (18.27), while the second term for regularization based on the KL-divergence is different:

$$\begin{aligned}
&\frac{\partial}{\partial \hat{\rho}_j} \left[\sum_{j=1}^{l} \rho \log \left(\frac{\rho}{\hat{\rho}_j} \right) + (1-\rho) \log \left(\frac{1-\rho}{1-\hat{\rho}_j} \right) \right] \frac{\partial \hat{\rho}_j}{\partial w_{jk}^h} \\
&= \left[\rho \frac{\partial}{\partial \hat{\rho}_j} (\log \rho - \log \hat{\rho}_j) + (1-\rho) \frac{\partial}{\partial \hat{\rho}_j} [(\log(1-\rho) - \log(1-\hat{\rho}_j))] \right] \frac{\partial}{\partial w_{ij}} g \left(\sum_{k=0}^{d} w_{jk}^h z_i \right) \\
&= \left(-\frac{\rho}{\hat{\rho}_j} + \frac{1-\rho}{1-\hat{\rho}_j} \right) g'(a_j) x_k
\end{aligned} \tag{18.47}$$

Now d_j^h in Eq. (18.27) for back propagation (defined in Eq. (18.28)) can be modified as below for the autoencoder:

$$d_j^h = \left[\left(\sum_{i=1}^{m} d_i^o w_{ij}^o \right) + \lambda \left(-\frac{\rho}{\hat{\rho}_j} + \frac{1-\rho}{1-\hat{\rho}_j} \right) \right] g'(a_j^h) \tag{18.48}$$

Using this modified d_j^h in step 6 of the algorithm for the propagation listed in the previous section, we get the algorithms for the autoencoder.

The Matlab code segment below is the essential part of the autoencoder algorithm, which takes the given dataset as input and generates the weights for both hidden and output layers as output. Here to achieve the desired sparsity of the hidden layer, the first approach mentioned above is used, i.e., out of all L hidden layer nodes, $La < L$ most active ones are kept while all others are set to zero.

Alternatively, the more elaborate approach based on KL-divergence can be used as well.

```
Wh=1-2*rand(L,d+1);                % hidden layer weights
Wo=1-2*rand(d,L+1);                % output layer weights
done=0;
it=0;
while ~done
    it=it+1;
    X=Xm;
    [d N]=size(X);
    I=randperm(N);                 % random order of N samples
    Ah=0;
    for n=1:N                      % for each samples
        x=X(:,I(n));               % input vector
        x1=[1;x];                  % augmented input vector
        ah=Wh*x1;                  % activation of hidden nodes
        z=g(ah);                   % output of hidden layer
        zsorted=sort(z,'descend'); % sort hidden layer output
        z=z.*(z>=zsorted(La));     % keep La most active nodes
        z=[1;z];                   % augment hidden layer output
        ao=Wo*z;                   % activation of output nodes
        xhat=g(ao);                % actual output of output layer
        do=(x-xhat).*dg(ao);       % delta of output layer
        Wo=Wo+eta*do*z';           % update output layer weights
        dh=(Wo(:,2:L+1)'*do).*dg(ah);  % delta of hidden layer
%        dh=((Wo(:,2:L+1)'*do)+lambda*(-r./rp+(1-r)./(1-rp))).*dg(ah);
        Wh=Wh+eta*dh*x1';          % update hidden layer weights
    end
end
```

Recall that in dimensionality reduction based on KLT, the transform matrix is the eigenvector matrix of either the covariance matrix $\mathbf{\Sigma}_x$ of the entire dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$, or the between-class scatter matrix $\mathbf{S}_B$ in the case of supervised classification. Similarly an autoencoder can also be trained based on either the entire dataset $\mathbf{X}$ for dimensionality reduction, or a dataset $\mathbf{X} = [\mathbf{m}_1, \cdots, \mathbf{m}_K]$ containing the mean vectors of some K classes (calculated based on labeled training set) for classification.

In the following examples, we compare the performances of both the autoencoder (trained on a dataset composed of only mean vectors of all classes) and the method PCA based on the between-class scatter matrix of the classes, when they are applied to reduce the dimensionality of the given dataset to be classified by the naive Bayes classifier.

Example 18.6 Both methods of autoencoder and PCA are applied to a dataset

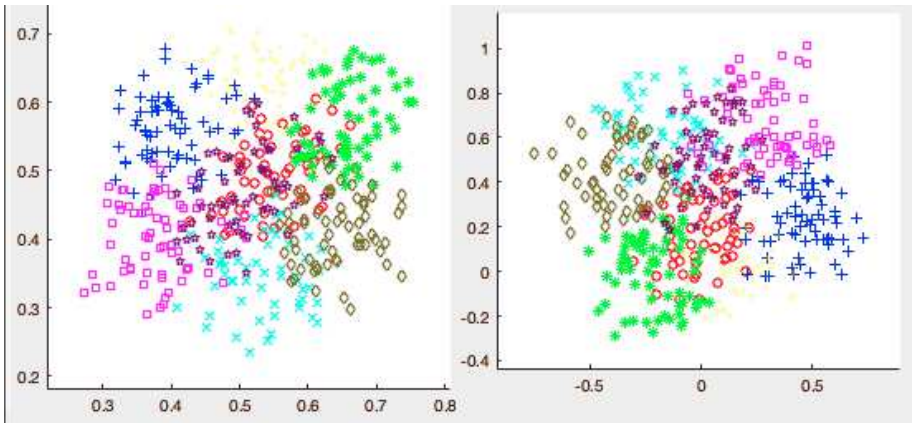


Figure 18.8 Autoencoder (left) vs. PCA (right) for Dimension Reduction for a 3-D Dataset

containing $K = 8$ classes each in one of the octants of the 3-D space, and then compared in terms of the separability and classification error rate in the resulting lower dimensional space. Note that in this case all three dimensions are equally important for representing the 8 classes, as also reflected by the eigenvalues 0.37%, 0.33%, 0.30% of the covariance matrix, indicating that roughly the same amount of information is contained in each of the three components of the data. We see that it is difficult to reduce the dimensionality without losing significant amount of information.

Fig. 18.8 shows the dataset after its dimensionality if reduced from 3 to 2 by autoencoder (left) and PCA (right) methods. by visual observation, We see that the data points are similarly distributed in the resulting 2-D space, but the class separability for the autoencoder seems to be slightly better than that by the PCA. This result can be confirmed quantitatively as shown in the first row of Table 18.49, where the separability $J_{B/T} = tr\left(\mathbf{S}_T^{-1}\mathbf{S}_B\right)$ (Eq. (9.24)) and the error rates for classification (by naive Bayes classifier) are listed to left and right of the slash symbol for the autoencoder and PCA methods, respectively. showing the autoencoder has a slightly high separability and lower error rate then the PCA method.

	Separability	Error rate
8-class Dataset	1.52/1.51	0.304/0.320
Iris Dataset	1.12/1.07	0.020/0.027

Example 18.7 The autoencoder and PCA methods are also compared when applied to the iris dataset containing three classes in the 4-D space. The resulting 2-D dataset is shown in Fig. 18.9, and the separabilities the classification error

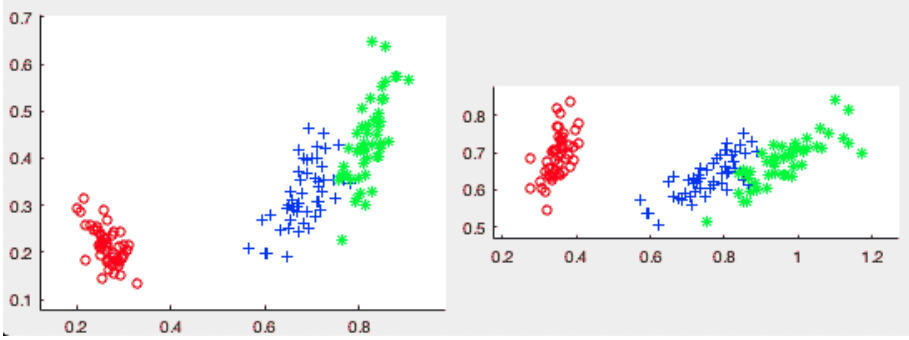


Figure 18.9 Autoencoder (left) vs. PCA (right) for Dimensionality Reduction for Iris dataset

rates of the two methods are compared in the second row of Table 18.49. We see again that the autoencoder has a little higher separability and lower error rate than the PCA method.

Based on the separabilities and classification error rates listed in Table 18.49, we see that for both datasets, the performance of the autoencoder (to the

Example 18.8 The autoencoder network is typically trained by all samples in the dataset. However, if these samples in a dataset are labeled to belong to one of a set of classes, the autoencoder can also be trained by these classes. Specifically in this example, the three classes in the iris dataset each represented by the average of all samples in the class are used to train the autoencoder. Fig. 18.10 shows the 4-D iris dataset mapped to 2-D (top) and 3-D (bottom) spaces by both the autoencoder (left) and KLT (right). It is interesting to observe that in the case of autoencoder (left), the data points representing the three classes are arranged into a line structure in both 2 and 3-D spaces.

Example 18.9 The same method is applied to the dataset of 10 handwritten digits from 0 to 9, each represented by 224 training samples in image form of 16×16 pixels, treated as a 256 dimensional vector. After the autoencoder network is fully trained based on the means of all 224 samples of each of the 10 classes, the dataset containing $224 \times 10 = 2240$ data samples in the l-dimensional space spanned by the hidden layer nodes that encode the 256-d samples is then visualized in 3-D space based the KLT transform, as shown in Fig. 18.11, where the dataset in the 256-D space is mapped to $l = 3, 4, 5, 6$ dimensional space by the autoencoder (with l hidden layer nodes) as well as the KLT method, and then further mapped down to 3-D by KLT for visualization.

Also, as shown in Fig. (18.12) for the plots of the separabilities of the ten classes in the lower dimensional space, the autoencoder and KLT method achieve comparable separabilities.

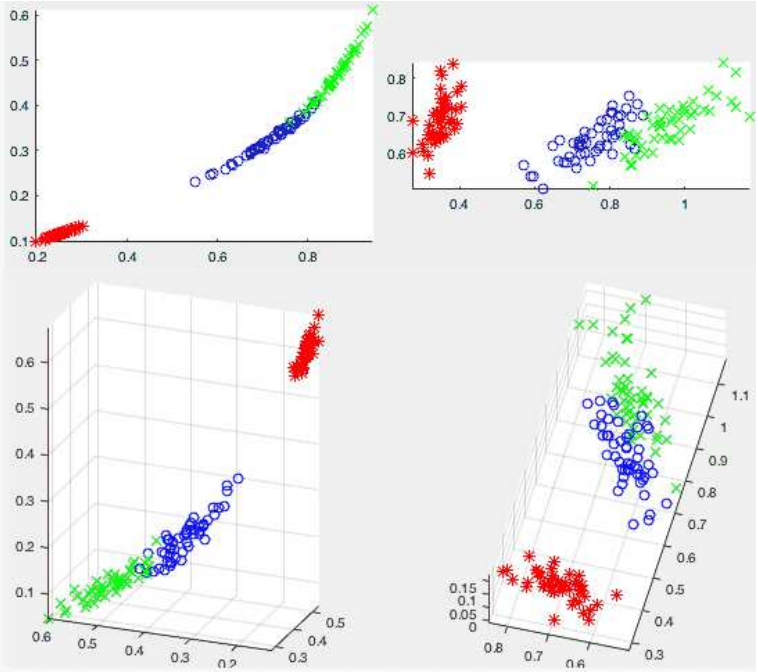


Figure 18.10 Autoencoder (Left) vs. KLT (right) for Dimension Reduction

18.4 Deep Learning

Deep learning, a method based on multi-layer feed forward network with multiple hidden layers, has gained popularity and caused a new wave of enthusiasm for artificial neural networks in the recent years, due heavily to the progress in computational capability as well as the latest development in new algorithms.

Conceptually, deep learning network can be simply understood as a network composed of multiple learning layers, including more than one hidden layer as well as the output layer. The perceptron network with a single learning layer and the three-layer network with the hidden and output learning layers discussed previously can be considered as special cases of deep learning. If we model a single learning layer by a nonlinear function that maps an input $\mathbf{x}$ to an output

$$\mathbf{y} = \mathbf{g}(\mathbf{w}^T\mathbf{x}) = \mathbf{f}(\mathbf{x}, \mathbf{w}) \tag{18.49}$$

than an L-layer deep learning network can simply be described by

$$\mathbf{y} = \mathbf{f}^{(L)}(\mathbf{f}^{(L-1)}(\cdots \mathbf{f}^{(1)}(\mathbf{x}, \mathbf{w}^{(1)}), \mathbf{w}^{(L-1)}), \mathbf{w}^{(L)}) = \mathbf{f}(\mathbf{x}, \mathbf{W}) \tag{18.50}$$

where $\mathbf{f}$ represents the cascade of all the L single-layer mapping functions $\mathbf{f}^{(1)}$ to $\mathbf{f}^{(L)}$, and $\mathbf{W} = [\mathbf{w}^{(1)}, \cdots, \mathbf{w}^{(L)}]$ is a matrix containing all weights in the network. When properly trained, such a multi-layer learning network will be able to model

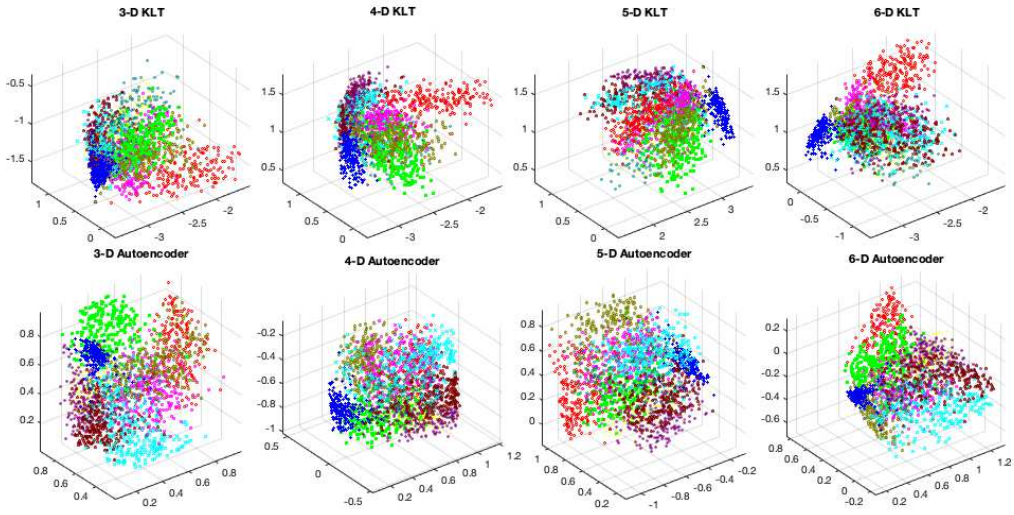


Figure 18.11 Dimensionality Reduction by Autoencoder and KLT

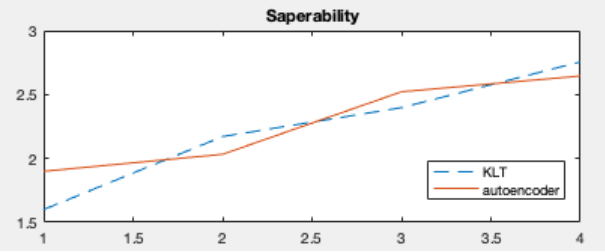


Figure 18.12 Separability in Lower Dimensional Space

some highly complicated and nonlinear relationship between input $\mathbf{x}$ and output $\mathbf{y}$.

When the multiple layers in a deep learning network are fully connected, i.e., a node in a layer gets inputs from all nodes in the previous layer weighted by $\mathbf{w}$, then the method of back propagation discussed previously can be used to update the weights based on gradient descent.

The basic structure of the multi-layer network has been inspired by the biology of the brain. The hierarchical structure of the network is composed of lower level layers where the neurons selectively respond to some simple local features in the data, while neurons in subsequent higher level layers selectively respond to progressively more complex and global patterns. Such a hierarchy is very similar to the two main visual pathways in the primate brain, known as the dorsal or "where" pathway and the ventral or "what" pathway, both formed by multiple levels of cortical areas starting from the primary visual cortex (V1),

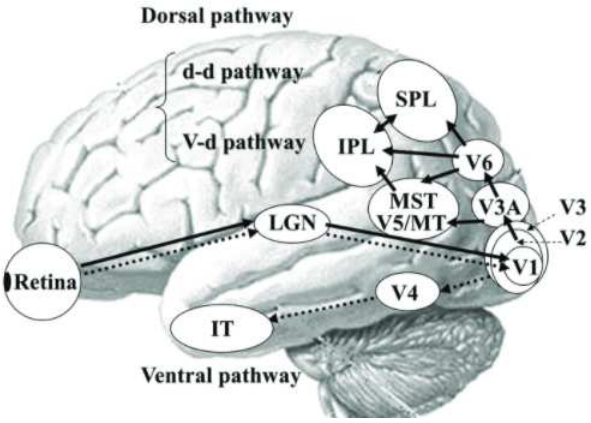


Figure 18.13 Visual Pathways

each composed of different types of neurons in terms of the size and specific type of the visual stimuli they are selectively responsive to.

Specifically, the hierarchy of ventral pathway known for visual object recognition is composed of neurons in the lower layer such as V1 where the simple cells with small receptive field selectively respond to edges of different orientations, and neurons with progressively larger receptive fields in the higher layers selectively respond to more complex features in larger areas of the visual field, such as neurons in the inferior temporal (IT) area sensitive to visual patterns such as image of faces. This visual pathway is shown in Figs. 18.13 ("Understanding the Pathophysiology of Alzheimer's Disease and Mild Cognitive Impairment: A Mini Review on fMRI and ERP Studies" *Neurology Research International*. 2012, Article ID 719056) and 18.14 ("Feature integration and object representations along the dorsal stream visual hierarchy",*Frontiers in Computational Neuroscience*, 05 August 2014, Vol 8 - 2014).

As an example of deep learning networks, the convolutional neural network (CNN) is hierarchical structure composed of a sequence of multiple 2-D layers for the purpose of image object recognition. This network can be considered to a large extent a model that mimics the ventral pathway of the visual cortex described above, in the sense that the nodes in the lower layers detect and extract local features in the visual data (mimicking the simple cells in V1), while the nodes in higher layers respond selectively to more global patterns represented in terms of the local features detected by the lower layers (mimicking the cells in IT).

Specifically, in the lower layers of the CNN, local image features such as edges are extracted by convolution, a basic technique commonly used in image processing. Mathematically, a 1-D continuous convolution in time domain is an operation

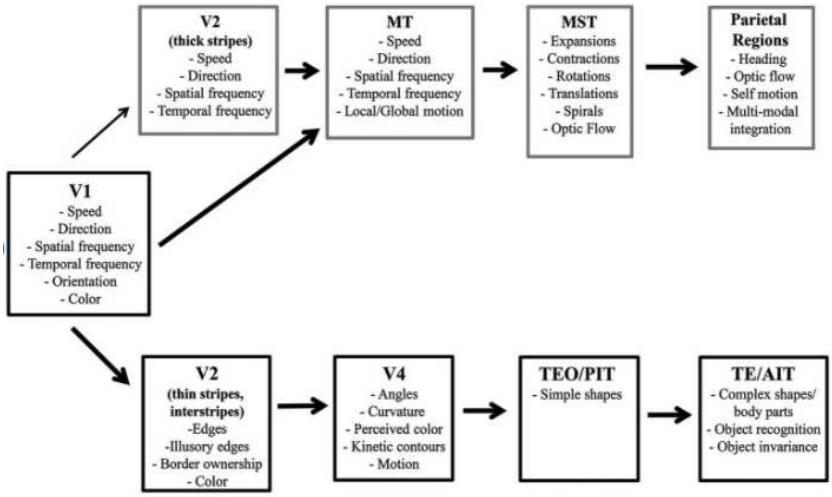


Figure 18.14 Specialization of Visual Cortical Areas

defined for two functions $f(t)$ and $g(t)$:

$$y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} x(t-\tau)h(\tau)d\tau = \int_{-\infty}^{\infty} h(t-\tau)x(\tau)d\tau \tag{18.51}$$

or a 2-D discrete convolution is spatial domain is an operation defined for two 2-D discrete functions $x[m, n]$ and $h[m, n]$:

$$y[m,\, n] = \sum_{i=-k}^{k} \sum_{j=-k}^{k} x[m-i,\, n-j]\; h[i,\, j] \tag{18.52}$$

where $h[m, n]$ is called the *convolution kernel* which typically is assumed to be central symmetric $h[-m, -n] = h[m,\, n]$. Such a convolution is sometimes also considered as a *filters*, by which a 2-D signal $x[m, n]$, typically a digital image, is filtered in some way (low-pass, high-pass, band-pass, etc.) to become $y[m, n]$. Fig. 18.15 illustrates a 2-D convolution with a 3×3 kernel.

We see that each pixel $y[m,\, n]$ of the resulting image y is the sum of 9 products. As the kernel slides over the input image x pixel by pixel and then row by row, all pixels of y are calculated.

For example, the four kernels below are can be used to detect edges in vertical and horizontal orientations:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}, \quad \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{18.53}$$

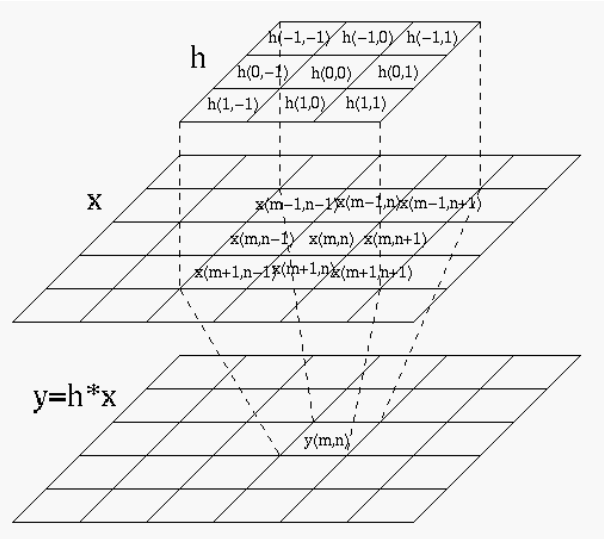


Figure 18.15 2-D Convolution

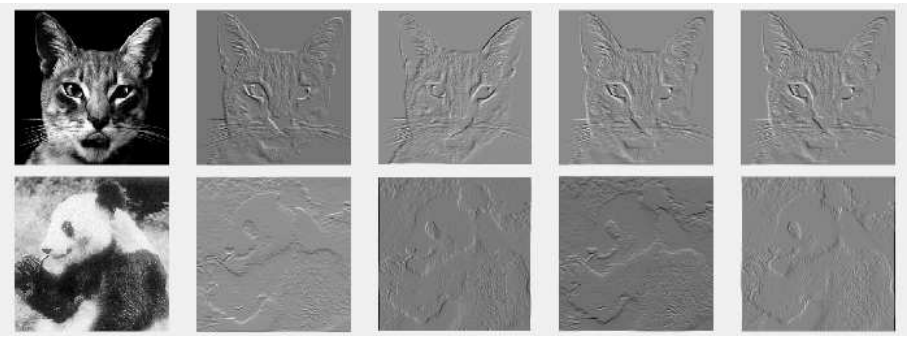


Figure 18.16 Directional Filtering

and the following four kernels are for detecting edges in two diagonal orientations:

$$\begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}, \quad \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix}, \quad \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{18.54}$$

Fig. 18.16 shows the effects of edge detection by convolution. The cat image is convolved with the kernels for in Eq. (18.53), while the panda image is convolved with kernels in Eq. (18.54).

We note that the convolution as a linear combination can be described by the general neural network operation $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{w})$ with the elements in the 2-D kernel $\mathbf{w}$ treated as the synaptic weights, through which the 2-D output $\mathbf{x}$ from a lower layer is fed forward into the next layer to produce its 2-D output $\mathbf{y}$.

These 3×3 kernels can be extended to larger kernels for detecting more complex features in a larger area of the visual field in higher layers of the network.

Specifically, the following operations take place sequentially along the multilayer pathway of the CNN hierarchy as shown in Fig. 18.17, each serving certain purposes:

- *Convolution:*

  Each node in the convolution layers is only locally connected to a set of neighboring nodes in a square patch in the previous layer. In particular, a node in the first layer take as input the pixel values inside its receptive fields, a subregion of the image. It then generates the convolution, the sum of the inputs weighted by the kernel components, and feed it to the next layer. At each position, multiple nodes with the same receptive field form a column in the depth dimension, all performing convolution but based on different kernel to extract different types of local features, such as edges in various orientations as shown above.
- *Padding:*

  Padding is needed for the output of convolution to maintain the same size as the input. For example, for a 5×5 kernel, the first pixel at the top-left corner of the output is in the position of the pixel in the 3th row and 3th column in the input, i.e., the output is 4 pixels smaller in both dimension than the input. To prevent this size reduction from one layer to the next, the input needs to be padded with additional rows and columns of pixels around its four borders (zeros or taking same value as the border pixels).
- *Stride*:

  Stride s is the number of pixels for the revolution kernel to shift in both dimensions in convolution. In regular convolution, $s = 1$, the size of the resulting image is the same as the input. But when $s > 1$, the kernel shifts s pixels from the center of one receptive field to next. In other words, stride is the distance between the receptive field centers of neighboring neurons. The size of the output will be $1/s$ of the original in both dimensions.
- *Pooling*:

  Pooling is a down-sampling method by which a local patch (e.g., 2×2) in the previous layer is represented by their average or maximum as a single value in the next layer called pooling layer, for dimension reduction that serves two purposes: (a) local shift and rotational invariance and (b) reduction of computational complexity.
- *Dilution or Dropout*:

  Certain fraction of nodes randomly selected in a layer is dropped by setting the corresponding weights to zero with a set probability, for the purpose of preventing overfitting, so that the network is less sensitive to certain detailed variations in the data due most likely to noise and thereby more generalizable.

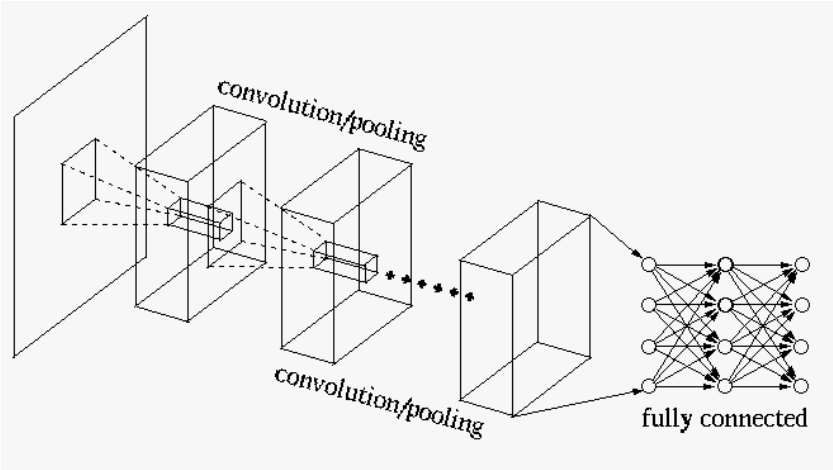


Figure 18.17 Hierarchical Convolutional Network

- In the last stage of the CNN, all nodes in a layer are fully connected to all nodes in the previous layer, they take as input the progressively larger and more complex features extracted in the previous, and learn to selectively respond to different image objects of interest based on error backpropagation algorithm, to achieve the ultimate goal of image objects recognition.
- *Weight decay*:

  As mentioned before, a multilayer neural network can be considered as a function $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{w})$ for modeling the relationship between $\mathbf{x}$ and $\mathbf{y}$, same as many supervised learning algorithms regression and classification. Similarly, including a penalty term proportional to $||\mathbf{w}||$ or its square in the objective function is an effective way to regulate the algorithm and avoid overfitting.

Problems

1. Implement the perceptron network as a linear classifier (based on both one-hot and binary encoding) for 3-D data generated by the function below. Show your results in confusion matrix and the error rate.

```
function [X,y]=Data
    d=3;                    % number of dimensions
    K=8;                    % number of classes
    Nk(1:K)=50;             % number of samples per class
    N=sum(Nk);              % total number of samples
    Means=[-1 -1 -1 -1 1 1 1 1; % mean vectors of K classes
           -1 -1 1 1 -1 -1 1 1;
```

```
        -1 1 -1 1 -1 1 -1 1];
    X=[];
    y=[];
    s=0.22;
    for k=1:K               % for each of the K classes
        Xk=Means(:,k)+s*randn(d,Nk(k));
        y=[y repelem(k,Nk(k))]
        X=[X Xk];
    end
end
```

2. Implement kernel perceptron network (based on radial basis kernel) and apply it to the classification of the iris dataset. Use both onehot and binary encoding methods. Use half of the data for training and the other half for testing. Show your results in confusion matrix and error rate.
3. Repeat the previous problem using to the handwritten digit dataset.
4. Implement the back propagation network and apply it to the handwritten digit dataset based on all $d = 256$ dimensions of the data. Use both one-hot (10 output nodes each for one class) and binary encoding (4 output nodes to encode 10 classes) labeling. Use half of the data for training and the other half for testing. Show your results in confusion matrix and error rate.

   Then use the KLT method based on the between-class scatter matrix $\mathbf{S}_B$ to reduce the data dimensionality while keeping 99% of the signal energy, and then repeat the classification task above.
5. Implement autoencoder for dimensionality reduction and apply it to the iris dataset. Visualize and classify the data in the resulting lower dimensional space (both 2-D and 3-D space). Then compare the separability $J_{B/T}$ and classification error rate with that obtained based on the PCA method.
6. Repeat the problem above using the handwritten digit dataset.

19 Competition-Based Networks

19.1 Competitive Learning Network

Competitive learning is a neural network algorithm for unsupervised clustering, similar to the K-means algorithm considered previously. The competitive learning takes place in a two-layer network, composed of an input layer of d nodes that receives an input vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ as a point in the d-dimensional feature space, and an output layer of K nodes that produces an output pattern $\mathbf{y} = [y_1, \cdots, y_K]^T$ representing the clusters of interest. Each input variable x_i may take either a continuous real value or a binary value depending on the specific application, the outputs $y_i \in \{0,\ 1\}$ are binary, of which only one is 1 while all others are 0 (one-hot), as the result of a winner-take-all competition based on their activation values.

In each iteration of the learning process, a pattern $\mathbf{x}$ randomly chosen from the unlabeled dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ is presented to the input layer of the network, while each node of the output layer gets an activation value, a linear combination of all such d inputs, i.e., the inner product of its weight vector $\mathbf{w}_i$ and the input vector $\mathbf{x}$:

$$a_k = \sum_{j=1}^{d} w_{ij} x_j = \mathbf{w}_k^T \mathbf{x} = ||\mathbf{w}_k||\ ||\mathbf{x}||\ \cos\theta \quad (k = 1, \cdots, K) \tag{19.1}$$

where θ is the angle between vectors $\mathbf{w}_k$ and $\mathbf{x}$ in the d-dimensional feature space. If the data vectors are normalized with unit length $||\mathbf{x}||^2 = 1$, they are points on the unit hypersphere in the space. Moreover, if the weight vectors $\mathbf{w}_k$ is also normalized with $||\mathbf{w}_k|| = 1$, then their inner product above is only affected by the angle θ between them. In practice, the normalization of both the data vector $\mathbf{x}$ and the weight vector $\mathbf{w}$ is optional.

The output of the network is determined by a winner-take-all competition. The node in the output layer that is maximally activated outputs 1 while all others output 0:

$$y_i = \begin{cases} 1 & \text{if } a_i = \mathbf{w}_i^T \mathbf{x} = \max_j \mathbf{w}_j^T \mathbf{x} \\ 0 & \text{otherwise} \end{cases}, \quad (i = 1, \cdots, K) \tag{19.2}$$

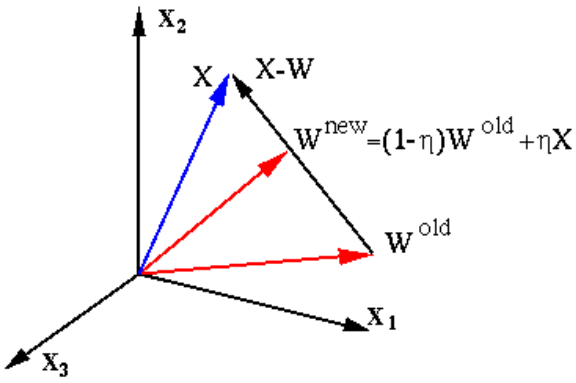


Figure 19.1 Modification of Weight Vector of the Winner

The inner product $\mathbf{w}_k^T\mathbf{x}$ is closely related to the Euclidean distance squared:

$$\begin{aligned} d^2(\mathbf{w}_k,\mathbf{x}) &= ||\mathbf{w}_k-\mathbf{x}||^2 = (\mathbf{w}_k-\mathbf{x})^T(\mathbf{w}_k-\mathbf{x}) = \mathbf{w}_k^T\mathbf{w} - 2\mathbf{w}_k^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\ &= ||\mathbf{w}_k||^2 + ||\mathbf{x}||^2 - 2\mathbf{w}_k^T\mathbf{x} = ||\mathbf{w}_k||^2 + ||\mathbf{x}||^2 - 2||\mathbf{w}_k||\,||\mathbf{x}||\,\cos\theta \end{aligned} \tag{19.3}$$

If both $\mathbf{x}$ and $\mathbf{w}_k$ are normalized, i.e., they becoem points on a unit sphere in the d-dimensional feature space, then the winning node with the maximum activation $a_k = \mathbf{w}_k^T\mathbf{x}$ also has the minimum Euclidean distance $||\mathbf{w}_k - \mathbf{x}||$ and angular distance θ. The competition above can therefore be expressed as the following:

$$y_k = \begin{cases} 1 & \text{if } ||\mathbf{w}_k-\mathbf{x}|| = \min_j ||\mathbf{w}_j-\mathbf{x}|| \\ 0 & \text{otherwise} \end{cases}, \qquad (i=1,\cdots,K) \tag{19.4}$$

Now the competitive learning rule can be expressed as below based on y_k:

$$\mathbf{w}_k^{new} = \mathbf{w}_k^{old} + y_k\,\eta\,(\mathbf{x}-\mathbf{w}_k^{old}) = \begin{cases} (1-\eta)\mathbf{w}_k^{old} + \eta\mathbf{x} & \text{if } y_k = 1 \\ \mathbf{w}_k^{old} & \text{if } y_k = 0 \end{cases} \tag{19.5}$$

i.e., only the winning node learns to respond to the current input by modifying its weight vector, while the weights of all other nodes remain the same. This competitive learning is illustrated in Fig. 19.1, where the modified weight vector of the winner $\mathbf{w}^{new}$ is between $\mathbf{x}$ and the old $\mathbf{w}^{old}$, i.e., the weight vector of the winner which is closest to the current input pattern $\mathbf{x}$ is pulled even closer to it, so that the node will be more likely to win and respond to the same input when it is presented again to the input of the network.

In the competitive learning rule, $0 < \eta < 1$ is the learning rate (step size), that is to be gradually reduced through the iterations from its initial value (e.g., $\eta_0 = 0.9$) toward zero, by a decay factor α. Due to the exponential decay of the learning rate, the iteration gradually converges to a stable state. The value of the decay factor (e.g., $\alpha = 0.99$) can be determined heuristically and experimentally depending on factors such as the size of the dataset.

Here are the iterative steps in the competitive learning:

- Step 0: Normalize all data vectors $\mathbf{x}$, initialize randomly the weight vectors $\mathbf{w}$ for the output nodes (e.g., any K randomly chosen samples from the dataset): $\mathbf{w}_1^{(0)}, \mathbf{w}_2^{(0)}, \cdots, \mathbf{w}_K^{(0)}$, set iteration index to zero $l = 1$;
- Step 1: Choose randomly an input pattern $\mathbf{x}$ from the dataset and calculate the activation for each of the output nodes:

$$a_k = \mathbf{w}_k^T \mathbf{x} \qquad (k = 1, \cdots, K) \tag{19.6}$$

- Step 2: find the index of the winning node $i = \arg\max_j a_j$, and update its weights:

$$\mathbf{w}_i^{(l+1)} = \mathbf{w}_i^{(l)} + \eta(\mathbf{x} - \mathbf{w}_i^{(l)}) \tag{19.7}$$

  Reduce learning rate $\eta \leftarrow \alpha\eta$, renormalize $\mathbf{w}_i^{(l+1)}$.
- Step 3: Terminate the iteration if the clustering result has gradually stabilized, when the learning rate η is reduced from its initial value η_0 to some small value (e.g., 0.1) and the weight vectors no longer change significantly. Otherwise $l \leftarrow l + 1$, go back to Step 1.

This iterative process can be more intuitively understood as shown in Fig. 19.2. Every time a sample $\mathbf{x}$ (a red dot) is presented to the input layer, one of the output nodes with its weight vector $\mathbf{w}$ (an blue x) closest to the current input $\mathbf{x}$ will become the winner and its weight vector is drawn even closer to $\mathbf{x}$. This process is carried out iteratively many times, each cluster of similar sample points in the space will draw one of the weight vectors towards its central area, and the corresponding output node will always win and output 1 whenever any member of the cluster is presented to the network in the future. In other words, after this unsupervised learning, the feature space is partitioned into K regions each corresponding to one of the K clusters, represented by one of the output nodes, whose weight vector is in the central area of the region. We see that this process is very similar to the K-means clustering algorithm, in which each cluster is represented by the mean vector of all of its members.

It is possible in some cases that the data samples are not distributed in such a way that they form a set of clearly separable clusters in the feature space. In the extreme case, they may even form a continuum. In such cases, a small number of output nodes (possibly even just one) may become frequent winners, while others become "dead nodes" if they never win and consequently never get the chance to modify their weight vectors. To avoid such meaningless outcome, a mechanism is needed to ensure that all nodes have some chance to win. This can be implemented by including an extra bias term in the learning rule:

$$a_k = \mathbf{w}_k^T \mathbf{x} + b_k \tag{19.8}$$

where b_k is the bias term proportional to the difference between the "fair share"

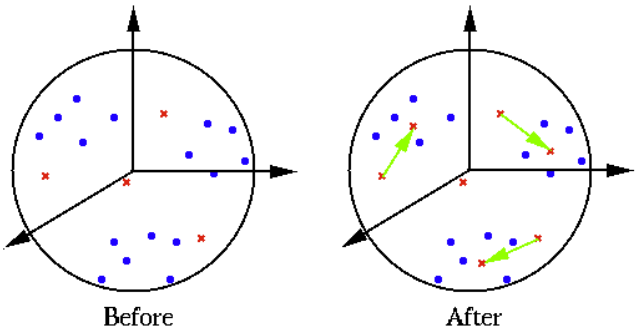


Figure 19.2 Weight Vectors Moving toward Centers of Clusters

of winning $1/K$ and the actual winning frequency:

$$b_k = c\left(\frac{1}{K} - \frac{\text{number of winnings of the kth node}}{\text{total number iterations so far}}\right) \tag{19.9}$$

The value of the bias term b_k for the kth node will change its winning frequency. If the node is winning more than its share $1/K$, then $b_k < 0$ and it becomes harder for it to win in the future. On the other hand, if the node rarely wins, $b_k > 0$ and its chance to win in the future will be increased. Here hyperparameter c is some scaling coefficient. The greater c, the more balanced the competition will be. It needs to be fine tuned based on the specific nature of the data being analyzed.

This process of competitive learning can also be viewed as a *vector quantization* process, by which the continuous vector space is quantized to become a set of K regions, called *Voronoi diagram (tessellation)*. Vector quantization can be used for data compression. A cluster of similar signals $\mathbf{x}$ in the vector space can all be approximately represented by the weight vector $\mathbf{w}$ of one of a small number of K output nodes in the neighborhood of $\mathbf{x}$, thereby the data size can be significantly reduced.

Same as in K-means clustering, competitive learning network also suffers from the problem that the ground truth number of clusters is typically unknown, and therefore it is difficult to determine the number of output nodes K of the network, which in the ideal case should be the same as the number of clusters in the dataset, so that each cluster can be represented by one of the nodes at the end of the clustering process. Similar to the case of K-means clustering, this issue can be addressed by carrying out the clustering process multiple times with different K values and checking to see which value best fits the data measured quantitatively by one of the criteria for the separability of the resulting cluster, such as those discussed in Section 15.1.

The Matlab code for the iteration loop of the algorithm is listed below.

```
[d N]=size(X);                % dataset containing N samples
```

```
b=zeros(1,K);                % bias terms for K clusters
freq=zeros(1,K);             % winning frequencies
eta=0.9;                     % initial learning rate
decay=0.99;                  % decay factor
it=0;
while eta>0.1                % main iterations
    it=it+1;                 % iteration counter
    W0=W;                    % initial weight vectors
    x=X(:,randi(N,1));       % randomly select an input sample
    dmin=inf;
    for k=1:K                % find winner among all output nodes
        d=norm(x-W(:,k))-b(k);
        if dmin>d
            dmin=d; m=k;     % mth node is the winner
        end
    end
    w=W(:,m)+eta*(x-W(:,m)); % modify winner's weights
    W(:,m)=w/norm(w);        % renormalize its weights (optional)
    share(m)=share(m)+1;     % update share for winner
    b=c*(1/K-share/it);      % modify biases for all nodes
    eta=eta*decay;           % reduce learning rate
end
```

Example 19.1 The competitive learning method is applied to a set of simulated 2-D data of four clusters. The network has $d = 2$ input nodes and $m = 4$ output nodes. The first 16 iterations are shown in Fig. 19.3, where open circles represent the data points, while the solid back squares represent the weight vectors. Also, the blue and red squares represent the weight vector of the winner before and after its modification, visualized by the straight line connecting the two squares. We see that the weight vectors randomly initialized are iteratively modified one at a time, and after these 16 iterations, they have each moved to the center of one of the four clusters. The separability of the clustering result measured by $tr(\mathbf{S}_T^{-1}\mathbf{S}_B)$ is 1.0. When the number of output nodes is reduced from the ground truth 4 to 3 and 2, the separability is also reduced to 0.76 and 0.46, respectively. When $K = 5$ output nodes are used, one of the four clusters is represented by two output nodes, indicating they can be subsequently merged.

Example 19.2 Fig. 19.4 shows the clustering results of a set of data points in a 3-D space, in terms of the weight vectors before (top) and after (bottom) of the competitive learning. The left column shows the 2-D space spanned by the first two principal components, while the right column shows the weight vectors in the original 3-D space. Again we see that the weight vectors move from their random initial positions to the centers of the eight clusters as the result of the clustering.

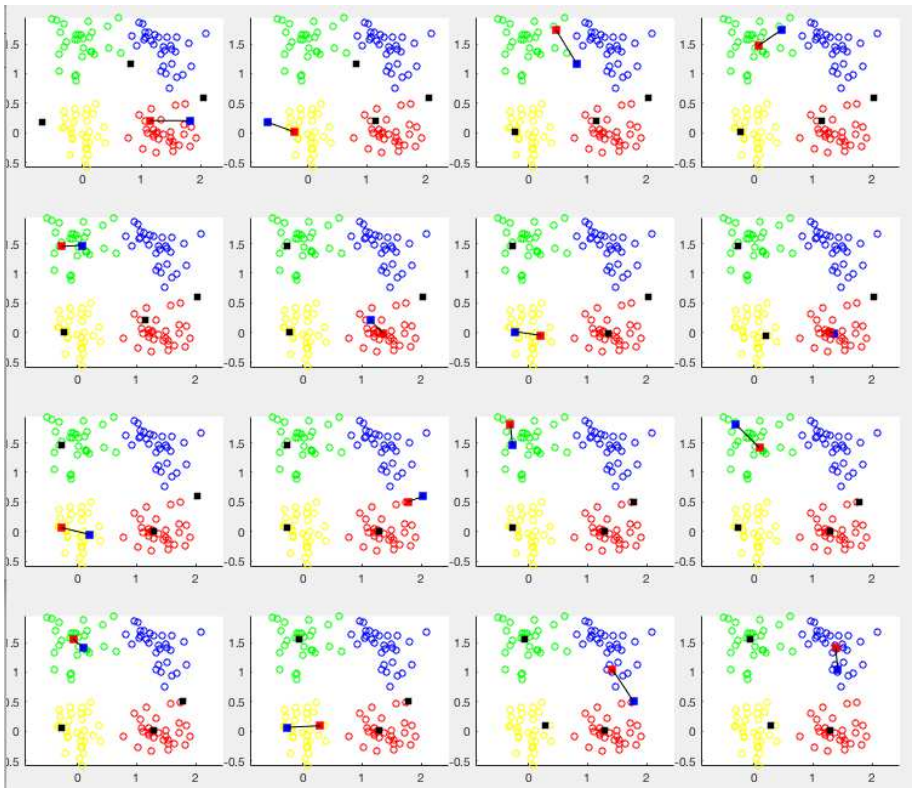


Figure 19.3 Iterative Learning of Competitive Network (2-D)

Example 19.3 The competitive learning algorithm is also applied to the handwritten digit dataset, and the clustering result is shown as the average of all samples in each cluster in 16×16 image form shown in Fig. 19.5.

19.2 Self-Organizing Map (SOM)

The *Self-organizing map (SOM)* is a two-layer unsupervised neural network learning algorithm that maps any input pattern presented to its input layer, a vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ in a d-dimensional feature space, to a set of output nodes that forms a low-dimensional space called *feature map*, typically a 2-D grid (lattice), although 1-D and 3-D spaces can also be used. This is shown in Fig. 19.6. A SOM can therefore be considered as a dimensionality reduction method, by which the dataset in the high-dimensional feature space can be visualized and the features and structures of the dataset can be discovered.

The learning rule of the SOM is similar to that of the competitive learning network considered previously in that all output nodes compete based on their activation levels upon the input pattern presented to input nodes of the network.

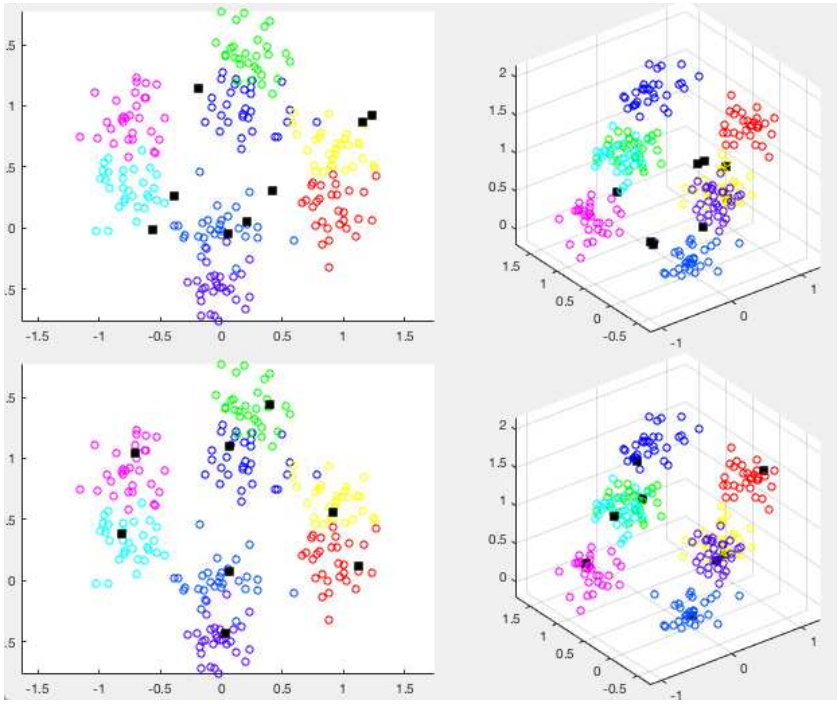


Figure 19.4 Example of Competitive Learning (3-D)



Figure 19.5 Clustering of Handwritten Digits by Competitive Learning

However, different from the competitive learning network where only the winning node gets to modify its weight vector and learn to respond to a cluster of similar patterns individually and independently, here in SOM network all output nodes in the neighborhood of the winning node get to modify their weight vectors and thereby learn to respond collectively to a cluster of similar input patterns. As the result, all output nodes are locked into a continuum feature map with smooth and gradual variation in responsive selectivity.

The SOM as a learning algorithm is motivated by the topographic organization and and columnar structures in the brain, in which signals from various visual or auditory sensory systems are topographically projected (mapped) to

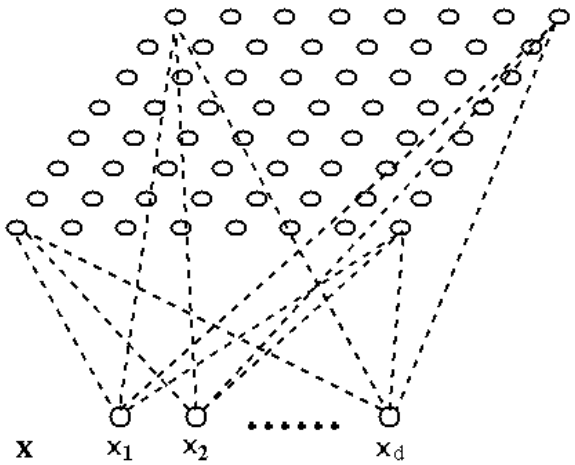


Figure 19.6 Self-Organizing Map

the corresponding cortical areas, where neighboring neurons in a local cortical area respond selectively to similar stimuli.

- *Retinotopic map:* Spatially adjacent stimuli in the visual field are represented and responded to by neurons in adjacent positions in the primary visual cortex (V1).
- *Tonotopic map:* Sound signals of different frequencies are mapped to the primary auditory cortex where neighboring neurons respond to similar frequencies.
- *Columnar architecture:* Visual signals for different edge orientations and different motion directions are mapped respectively to the primary visual area (V1) and the middle temporal area (V5 or MT), where neurons in the same column respond to the same edge orientations or similar motion directions, and neurons in neighboring columns respond to similar edge orientations or motion directions.

In a typical SOM network, the output nodes are organized in a 2-D feature map, and the competitive learning algorithm previously discussed is modified so that the learning takes place at not only the winning node in the output layer, but also a set of nodes in the neighborhood of the winner. The weight vectors of all such nodes are modified:

$$\mathbf{w}_i^{new} = \mathbf{w}_i^{old} + u_i \eta \left(\mathbf{x} - \mathbf{w}_i^{old} \right) \tag{19.10}$$

where u_i is a weighting function based on the Euclidean distance d_i from the ith node in the neighborhood to the winner in the 2-D feature map, such as a Gaussian function centered at the winner:

$$u_i = \exp\left(-\frac{d_i^2}{2\sigma^2} \right) \begin{cases} = 1 & \text{for winner with } d_i = 0 \\ < 1 & \text{for all other nodes} \end{cases} \tag{19.11}$$

where σ is a hyperparameter that controls the width of the Gaussian function. We see that all neurons in the neighborhood of the winner learn from the current input pattern in such a way that their weight vectors are modified to be closer to the input vector in the d-dimensional feature space, but they do so to different extents. Specially, for the winner itself with $d_i = 0$ and $u_k = 1$, the learning rate for the winner is maximally η, while all other nodes with $d_i > 0$ also modify their weights but with some lower rates $u_i\eta$. Those closer to the winner with smaller d_i and thereby greater u_i will learn more than those farther away.

Here are the specific steps of the SOM learning algorithm:

1. Initialize weights of a set of output nodes arranged in a 2-D map.
2. Choose randomly an input pattern vector $\mathbf{x}$ in the high-dimensional feature space and calculate the activation $y_i = \mathbf{w}_i^T\mathbf{x}$ for each of the output nodes.
3. Find the winning node with maximum activation y_k and update the weights of all nodes in its neighborhood by Eq. (19.10).
4. Go back to step 2 until the feature map is no longer changing or a set maximum number of iterations is reached.

In the training process, both σ controlling the size of the neighborhood and η for the learning rate will be gradually reduced.

The Matlab code segment listed below is for the essential part of the SOM algorithm.

```
[d N]=size(X);              % dimension and number of samples
W=rand(d,M,M);              % initialization of weights
for i=1:M
    for j=1:M
        w=reshape(W(:,i,j),[d 1]);
        W(i,j,:)=W(:,i,j)/norm(w); % normalize all weight vectors
    end
end
eta=0.9;                    % initial learning rate
sgm=M;                      % width of Gaussian neighborhood
decay=0.999;                % rate of decay
for it=1:nt                 % training iterations
    n=randi([1 N]);
    x=X(:,n);               % pick randomly an input
    amax=-inf;
    for i=1:M               % find the winner
        for j=1:M
            w=reshape(W(:,i,j),[d 1]);
            a=x'*w;         % activation as inner product
            if a>amax
                amax=a; wi=i; wj=j; % record winner so far
```

```
            end
        end
    end
    for i=1:M
        for j=1:M
            dist=(wi-i)^2+(wj-j)^2; % distance to winner
            c=exp(-dist/sgm);       % Gaussian weights
            w=reshape(W(:,i,j),[d 1]); % get a weight vector
            w=w+eta*c*(x-w);  % modify weights
            w=w/norm(w);      % normalize weight vectors
            W(:,i,j)=w;       % save modified weight vector
        end
    end
    eta=eta*decay;            % reduce learning rate
    sgm=sgm*decay;            % reduce width of Gaussian
end
```

Example 19.4 This example of "self-organizing map" is how the SOM algorithm got its name. The output nodes organized into a 2-D map learn to respond to the positions of a set of random points $\mathbf{x} = [x_1, x_2]^T$ in a $d = 2$ dimensional feature space. The weights of the output nodes are randomly initialized, and then modified iteratively during the learning process when the data points are repeatedly presented to the input of the network. Once the SOM has been trained, the output nodes close to each other in the 2-D feature map respond to a set of adjacent data points in the 2-D space, thereby forming a self-organized spatial map, a roughly regular grid.

The 2-D map of the output nodes is visualized as shown in Fig. 19.7, where the spatial position of each node is determined by the two components of its weight vector $\mathbf{w} = [w_1, w_2]^T$ treated as its coordinates, and each nodeis connected to its four neighbors in the 2-D grid to indicate the spatial structure of the feature map.

In this example, the competitive learning is based on the distance between the weight vector $\mathbf{w}_i$ and the current input $\mathbf{x}$, instead of their inner product $\mathbf{w}_i^T\mathbf{x}$. The node with the shortest distance $||\mathbf{w}_i - \mathbf{x}||$ becomes the winner, and its weight vector $\mathbf{w}$ is pulled even closer to the input, dragging along with it all its neighboring nodes.

The iterative modification of the weights is shown in Figs. 19.7 for the first 16 iterations and 19.8 for the subsequent iterations (in every 50 iterations). In each of the panels, the output node (blue dot) closest to the current input vector (red dot) becomes the winner and pulls the weight vectors of all neighboring nodes toward the input. This process converges to a configuration in which the output nodes form roughly a regular 2-D grid, a self-organized map in the 2-D feature space.

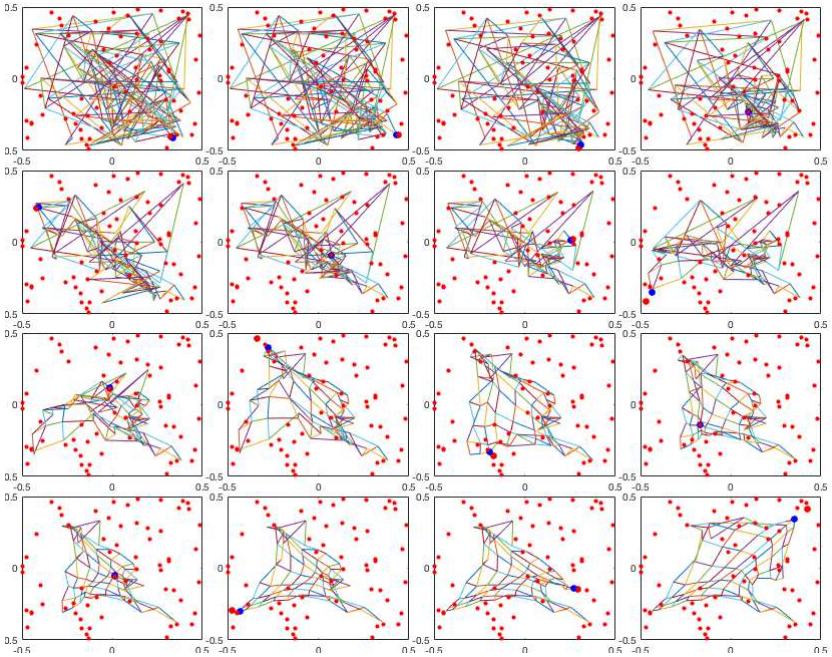


Figure 19.7 Iterative Learning of SOM (First 16 Iterations)

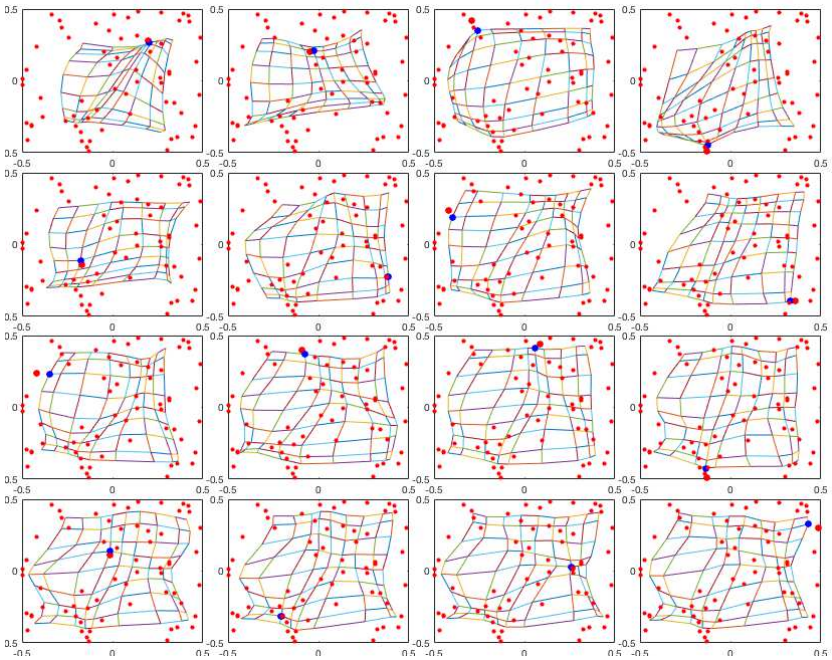


Figure 19.8 Iterative Learning of SOM (Subsequent Iterations)

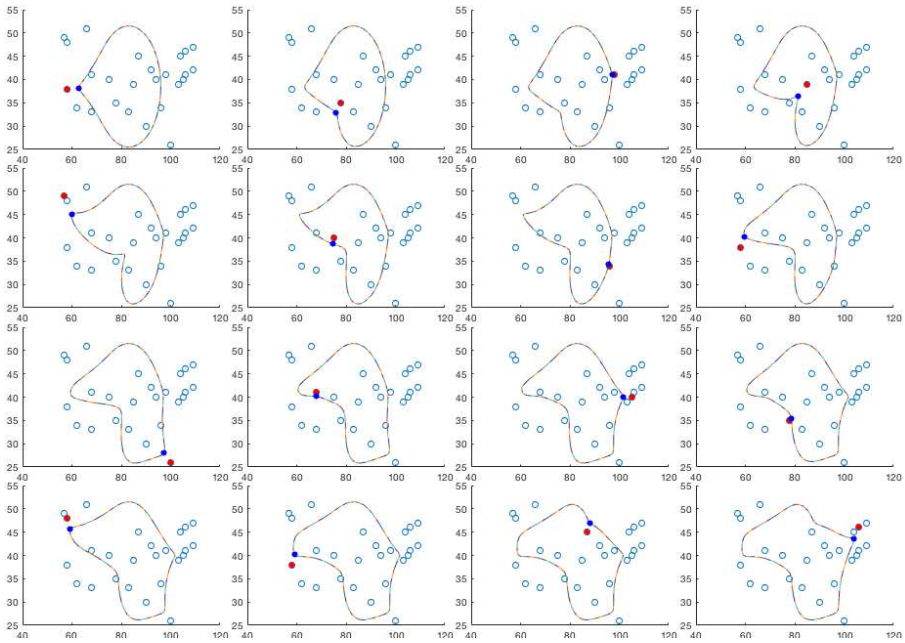


Figure 19.9 SOM Applied to the TSP (First 16 Iterations)

Example 19.5

In this example the SOM algorithm is applied to address the *traveling salesman problem (TSP)*. The SOM learns to respond to the locations of a set of 25 North America cities represented by their coordinates (longitudes and latitudes) and thereby approximate the shortest path through all these cities. The initial locations of the weight vector are arranged as a 1-D loop around the cities. And the nodes of the SOM gradually learn to modify their weights based on the city locations, so that they move toward the cities nearby to eventually form a reasonably short path through all these cities. The final result is unaffected by how the weights are initialized.

This learning process is illustrated in Figs. 19.9 for the first 16 iteration and 19.10 for the subsequent iterations (in every 50 iterations), where the output nodes are represented by their weight vectors as points along the curve in the 2-D map, together with the locations of the cities as the input to the SOM one at a time.

The resulting configuration of the output nodes is not guaranteed to be the desired shortest path through all cities, but it is should be reasonably close to such a solution.

Example 19.6 In this example, a SOM network learns to respond to different colors, as points in the color space spanned by the three primary colors red, green

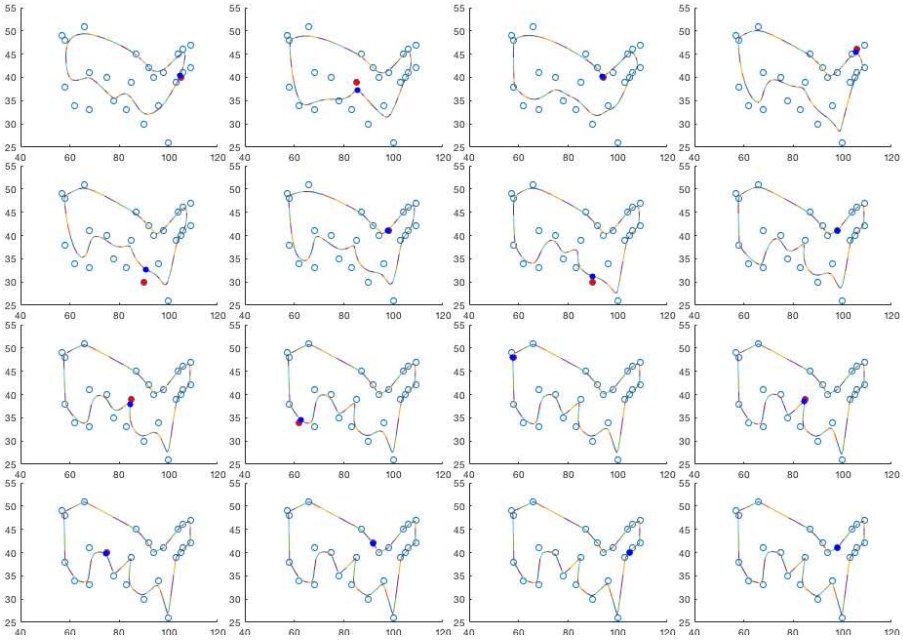


Figure 19.10 SOM Applied to the TSP (Subsequent Iterations)

and blue (RGB). Each node in the 2-D array of the output layer is connected to the three input nodes taking a 3-D vector for a color sample $\mathbf{x}$, and generates the inner product $\mathbf{w}^T\mathbf{x}$ of the weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$ as the activation. During training, the points in a 3-D regular grid of the color space are randomly selected as the training samples, and the weights of the SOM are gradually modified so that eventually each output node learns to respond most strongly to a specific color. Here all training vectors are normalized so that they have unity intensity, but their hue and saturation are determined by the ratios of the three primaries. Similarly, the weight vectors are also renormalized after each iteration in training. As $||\mathbf{w}|| = ||\mathbf{x}|| = 1$, the activation $\mathbf{w}^T\mathbf{x} = ||\mathbf{w}||\,||\mathbf{x}||\cos\theta = \cos\theta$ is maximized if the angle between θ between $\mathbf{w}$ and $\mathbf{x}$ is minimized, i.e., the weight vector matches the input color vector. Now the favored color of each output node can be shown by color coding the components of its weight vector as RGB, and the SOM can be visualized by the favored colors of all of its output nodes, as shown in Fig. 19.11. Here the left panel shows the SOM when all weights are randomly initialized, and middle panel shows the fully trained SOM with neighboring nodes responding to similar colors. The right panel further shows all the winners during the learning process, encoded by the input colors they win, while all other nodes that never win (but still learn to respond to certain colors along with their neighboring winners) are colored black. We see that the winners seem to form some structural patterns.

When the size of the SOM is progressively increased, more sophisticated web-

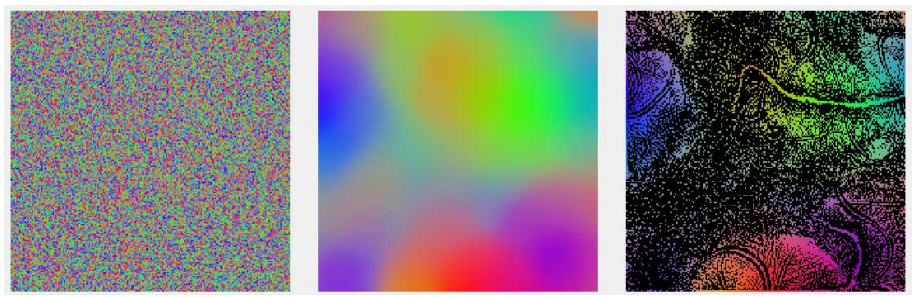


Figure 19.11 Colors Learned by a SOM

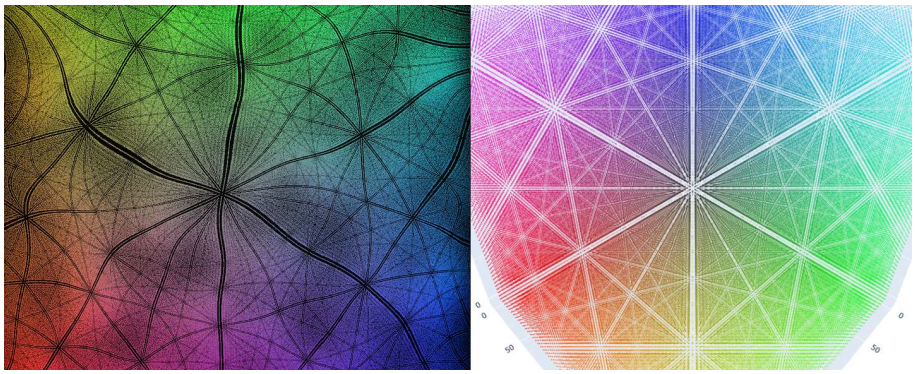


Figure 19.12 Emergent Patterns in SOM (left) and Perspective Projection of the Color Grid (right)

like patterns start to emerge, as shown in the left panel of Fig. 19.12 for a SOM of size 1000×1000. Such emergent patterns closely resemble the *perspective* projection of a regular grid in the 3-D color space onto an arbitrary 2-D plane, as shown in the right panel for a zoomed-in view of such a projection. We therefore see that the SOM network actually has learned to mimic the perspective projection of the training samples in the 3-D grid in the color space onto the 2-D output array. However, in an *orthographic* projection of the 3-D space as shown in Fig. 19.13 no such patterns emerge. An interesting question is why the SOM learns to mimic the perspective projection instead of a orthographic projection, while there is no specific mechanism that distinguish these two types of projections in the learning process.

Problems

1. Implement competitive learning network for clustering and apply it to the iris dataset. Show the result in confusion matrix and error rate. Also visualize

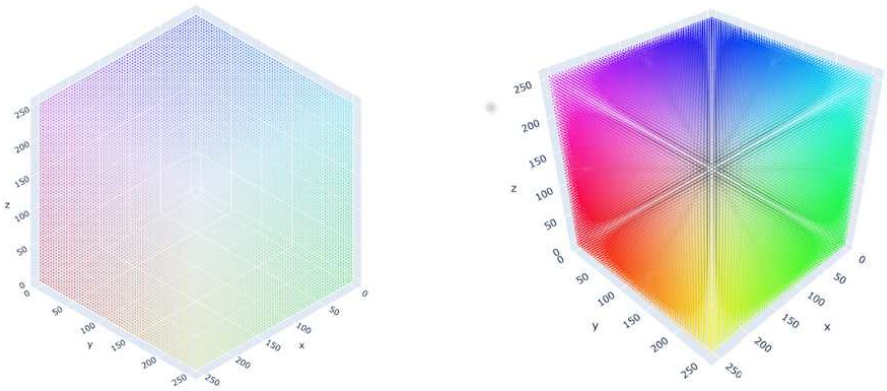


Figure 19.13 Orthographic and Perspective Projections of Color Grid
Orthographic (left) and perspective (right) projections of the color grid

your result in 2-D and 3-D spaces with all samples color coded according to (1) the cluster they are assinged to, and (2) their ground truth labeling.

2. Repeat the previous problem but based on the handwritten digit dataset. Also visualize the mean vectors of the $K = 10$ clusters converted back into 16×16 images to see what they actually represent.

   Compare all these results to those obtained by the K-means algorithm in the homework in Chapter 15.

3. Use the SOM network to learn different colors each represented by a 3-D vector for the three primary colors red, green, and blue. When the SOM has converged, show the resulting SOM by color (RGB) coding the output nodes in the 2-D array by their three weights, same as in Example 19.6.

   If interested, further show the SOM map but containing only those winning nodes during the training process. All other nodes that have never won during training will remain black. Try some large size SOM (e.g., 500×500) and large number of training samples (e.g., 10^3, i.e., 10 intensity levels in the range between 0 and 255 for each of the primary colors). Observe to see if any patterns show up in the map.

4. Use the SOM network to solve the traveling salesman problem for the major North American cities listed below together with their latitude and longitude. (More cities can be found at: ttp://www.infoplease.com/ipa/A0001796.html.)

	Lat.		*Long.*	
City	*minute*	*second*	*minute*	*second*
LosAngeles	34	3	118	15
SanFrancisco	37	47	122	26
NewYork	40	47	73	58
Chicago	41	50	87	37
Boston	42	21	71	5
Washington, D.C.	38	53	77	02
Miami	25	46	80	12
Minneapolis	44	59	93	14
NewOrleans	29	57	90	4
Seattle	47	37	122	20
Dallas	32	46	96	46
Cleveland	41	28	81	37
SaltLakeCity	40	46	111	54
Atlanta	33	45	84	23
Denver	39	45	105	0
KansasCity	39	6	94	35
Phoenix	33	29	112	4

Part VI

Reinforcement Learning

Part VI contains only one chapter to discuss the *reinforcement learning (RL)*, which is considered as one of the three basic machine learning paradigms, alongside supervised learning (e.g., regression in Part II and classification in Part IV), and unsupervised learning (e.g., clustering in Chapter 15). The goal of RL is for the algorithm, a software *agent*, to learn to make a sequence of decisions, called *policy*, in terms of the best *actions* to take for a specific task in a given *environment*, for the purpose of receiving the maximum cumulative *rewards* from the environment.

Unlike supervised learning for either regression or classification based on the training set composed of labeled data samples, RL is a learning method without any examples of optimal behaviors. Instead, RL learns on its own without relying on any labeled training data informing the agent explicitly what the correct responses are. On the other hand, Unlike unsupervised methods, which learns passively from the given dataset, RL is a trial-and-error learning method, in which the agent actively interacts with and learns from the given environment.

RL as a sequential method in the sense that the learning depends on the dynamics of the environment, which is modeled by a *Markov decision process (MDP)*, a stochastic system of multiple states with probabilistic state transitions and rewards. Starting from an initial state, the agent goes through a sequence of states to eventually reach a destination state. If the MDP model of the environment is known in terms of the state transition and reward probabilities, then the agent only needs to determine the optimal policy in terms of what action to take at each state in the sequence to transit to the next, so that the accumulated rewards can be maximized as the consequence of its actions. The task in this *model-based* case is called *planning*.

However, if the MDP of the environment is unknown as in the case of most real world problems, the agent needs to learn the MDP environment by repeatedly running the dynamic process while following some initial policy, and gradually evaluate the received rewards and improve the policy to eventually reach optimality. Such a process of learning the environment by interacting with it is called a *Monte Carlo (MC)* method and the task in this *model-free* case is called *control*.

Depending on the action a taken by the agent in the state s it is current in, the system transits to the next state s', one of the states available at s, according to a transition probability distribution $p(s'|s,a)$. The goal of RL is to choose the optimal action to maximize the long-term expected cumulative reward. This is typically realized by the method of *dynamic programming (DP)*, a class of algorithms that seeks to simplify a complex problem by breaking it up into a set of sub-problems to be solved recursively. The process is illustrated in Fig. 19.14 showing the t-th step of the process, where the agent makes a decision in terms of what action a_t to take in the current state $s_t = s$ by following certain policy $\pi(a|s)$, based on which the environment makes the corresponding transition to the next state $s_{t+1} = s'$ according to the transition probability $p(s'|s,a)$ conditioned on the current state $s_t = s$ and the agent's action a_t, and

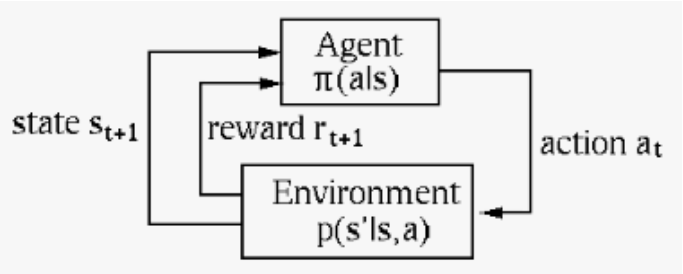


Figure 19.14 Reinforcement Learning: Agent and Its Environment

provides a reward r_{t+1}. We will consider the specific details of the DP algorithms in the subsequent sections.

20 Introduction to Reinforcement Learning

20.1 Markov Decision Process

20.1.1 Markov Chain

The mathematical framework for reinforcement learning takes the form of a *Markov decision process (MDP)*, based on the more basic concept of a *Markov process* or *Markov chain*, which is a stochastic model of a system that can be characterized by two types of parameters: (a) the set $S = \{s^1, \cdots, s^N\}$ containing all possible states of the system, and (b) the transition probability $P_{mn} = P(s^n|s^m)$ that models the dynamics of the system in terms of its state transition from the current state $s_t = s^m$ to the next state $s_{t+1} = s^n$ for all $m, n = 1, \cdots, N$. Here s^n denotes the nth state out of all $N = |S|$ states of the system (not s to the nth power), while s_t denotes the state at the t-th step of the process.

The system is assumed to be memoryless, i.e., its future depends on the current state s_t but is independent of the past history in terms of all previous states:

$$P(s_{t+1}|s_t, s_{t-1}, \cdots, s_0) = P(s_{t+1}|s_t) \tag{20.1}$$

For example, the motion of a helicopter can be modeled by a Markov process if its state is described by its linear and angular position and velocity, as its future position and velocity are completely determined by its present state. However it cannot be modeled by a Markov process, if the state is only described by its current position, while its velocity depends also on its positions in the past.

All transition probabilities can be organized as an $N \times N$ state transition matrix:

$$\mathbf{P} = \begin{bmatrix} P_{11} & \cdots & P_{1N} \\ \vdots & \ddots & \vdots \\ P_{N1} & \cdots & P_{NN} \end{bmatrix} \tag{20.2}$$

where $P(s^n|s^m)$ is the probability of the transition from state s_m to the state s_n. Since at any state s_m, the next state has to be one of the N possible states, we have:

$$\sum_{n=1}^{N} P_{mn} = \sum_{n=1}^{N} P(s^n|s^m) = 1, \qquad (m = 1, \cdots, N) \tag{20.3}$$

Any matrix satisfying this property is called a *stochastic matrix*. In the following

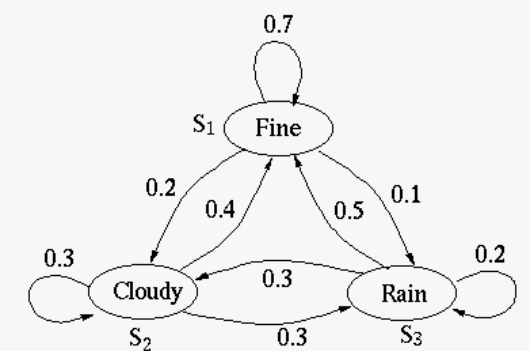


Figure 20.1 Stochastic matrix for LA Weather

discussion, we will concentrate mostly on the general transition from the current state $s_t = s$ to the next state $s_{t+1} = s'$ with transition probability denoted by $P(s_{t+1} = s'|s_t = s) = P_{ss'}$. For example, the transitions of the weather in Los Angeles (a lot of fine days!) are shown in Fig. (20.1), with the stochastic matrix shown in Eq. (20.4).

$$\mathbf{P} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.4 & 0.3 & 0.3 \\ 0.5 & 0.2 & 0.3 \end{bmatrix} \tag{20.4}$$

20.1.2 Markov Reward Process

A *Markov reward process (MRP)* is an extension of the Markov chain that also includes a reward associated with each of the states. The goal is for an agent to collect the maximum amount of rewards while going through the states of a dynamic process of the system. An MRP can be described by four sets of parameters, including the set of all rewards R, a *discount factor* γ, as well as the set of states S, and the state transition probabilities represented by P for a Markov chain. An MRP can therefore be represented by a quadruple (4-tuple) $\langle S, P, R, \gamma \rangle$.

Here the *reward* R is a feedback signal to the agent at each time step of the sequence of state transitions, indicating how well it is doing with respect to the overall task. The reward can be negative, as a penalty, for situations to be avoided. The performance of the agent is measured by the sum of rewards accumulated over all time steps of the Markov process, weighted by the discount factor $\gamma \in [0,\ 1]$ that discounts future rewards. If γ is close to 0, the immediate reward is emphasized, i.e., the policy is short-sighted or greedy; while if γ is close to 1, the rewards in the future steps will be almost as valuable as immediate ones, i.e., the policy is far-sighted.

The sequence of state transitions of an MRP from an initial state s_0 to a

terminal state s_T is called an *episode*, and the number of time steps T taken by the agent is called the *horizon*, which can be either finite or infinite.

Both the reward r and the next state s' at any given current state s are random variables with joint probability $p(s', r|s) = P(s_{t+1} = s', r_{t+1} = r|s_t = s)$, based on which both the state transition and reward probabilities can be found by marginalizing r and s' respectively:

$$p(s'|s) = P(s_{t+1} = s'|s_t = s) = \sum_{r \in R} p(s', r|s), \qquad P(r|s) = \sum_{s' \in S} p(s', r|s) \tag{20.5}$$

Conventionally the reward r received after arriving at state s_t is denoted by r_{t+1}, instead of r_t. The index $s' \in S$ for the summation over all states will be abbreviated to s' in the subsequent discussion. The expected reward in state s is

$$r(s) = E[r_{r+1}|s_t = s] = \sum_r rP(r|s) = \sum_r r \sum_{s'} P(s', r|s) \tag{20.6}$$

The *return* G_t in step $s = s_t$ is defined as the accumulated reward, the sum of the immediate reward after arriving at the current state $s = s_t$ and all delayed rewards in the future states up to and including reward r_T at the terminal state s_T at the end of the episode, discounted by γ:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} \tag{20.7}$$

The *state value* function $v(s)$ in state $s = s_t$ is defined as the expected return $E[G_t]$, which can be expressed recursively in terms of the values $v(s')$ of all possible next states $s' = s_{t+1}$:

$$\begin{aligned} v(s) &= E[\, G_t|s_t = s\,] = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots |s_t = s] \\ &= E[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \cdots)|s_t = s] \\ &= E[r_{t+1}|s_t = s] + \gamma E[G_{t+1}|s_{t+1} = s',\, s_t = s] \\ &= r(s) + \gamma \sum_{s'} P(s'|s)v(s') \end{aligned} \tag{20.8}$$

where $r(s)$ is given in Eq. (20.6). The value of the terminal state s_T at the end of an episode is zero, as there will be no next state and therefore no more future reward.

This equation is called the *Bellman equation*, by which the value $v(s)$ at current state s is expressed recursively in terms of the immediate reward $r(s)$ and the values $v(s')$ of all possible next states, without explicitly invoking all future rewards. Based on such a *bootstrapping* approach, the multi-step MRP problem is expressed recursively as a single-step subproblem concerning only one state transition from s to s', so that the current state value can be found from all future ones by induction. For this reason, the Bellman equation plays an essential role in all reinforcement learning algorithms to be considered below.

The Bellman equation in Eq. (20.8) holds for all N states $s^1, \cdots, s^N \in S$, and the resulting N equations can be expressed in vector form as:

$$\begin{aligned}\mathbf{v} = \begin{bmatrix} v(s^1) \\ \vdots \\ v(s^N) \end{bmatrix} &= \begin{bmatrix} r(s^1) \\ \vdots \\ r(s^N) \end{bmatrix} + \gamma \begin{bmatrix} P(s^1|s^1) & \cdots & P(s^N|s^1) \\ \vdots & \ddots & \vdots \\ P(s^1|s^N) & \cdots & P(s^N|s^N) \end{bmatrix} \begin{bmatrix} v(s^1) \\ \vdots \\ v(s^N) \end{bmatrix} \\ &= \mathbf{r} + \gamma \mathbf{P}\mathbf{v} \end{aligned} \tag{20.9}$$

where $\mathbf{P}$ is state transition matrix previously defined in Eq. (20.2), a stochastic matrix. Now the Bellman equation can be solved to get the value functions at all N states:

$$\mathbf{v} = (\mathbf{I} - \gamma \mathbf{P})^{-1} \mathbf{r} \tag{20.10}$$

Such a solution exists because matrix $\mathbf{I} - \gamma \mathbf{P}$ is invertible. To see this we note that all eigenvalues of the stochastic matrix $\mathbf{P}$ are no greater than 1: $|\lambda| \leq 1$ (Section A.3.1), and all eigenvalues of $\gamma \mathbf{P}$ are smaller than 1: $\gamma\lambda < 1$, i.e., all eigenvalues of $\mathbf{I} - \gamma \mathbf{P}$ are greater than zero: $1 - \gamma\lambda > 0$. Consequently the determinant $\det(\mathbf{I} - \gamma \mathbf{P})$ of this coefficient matrix, as the product of all its nonzero eigenvalues, is non-zero and therefore matrix $\det(\mathbf{I} - \gamma \mathbf{P})$ is invertible.

Alternatively, the Bellman equation in Eq. (20.8) can also be solved iteratively based on dynamic programming, which solves a multi-stage planning problem by backward induction and finds the value function recursively. We first rewrite the Bellman equation as

$$\mathbf{v} = \mathbf{r} + \gamma \mathbf{P}\mathbf{v} = B(\mathbf{v}) \tag{20.11}$$

where $B(\mathbf{v}) = \mathbf{r} + \gamma \mathbf{P}\mathbf{v}$ is defined as a vector-valued function (an operator) applied to the vector argument $\mathbf{v}$.

We can show that $B(\mathbf{v})$ is a contraction mapping first introduced in Section 1.3, that satisfies:

$$\begin{aligned} ||B(\mathbf{v}_i) - B(\mathbf{v}_j)|| &= ||\mathbf{r} + \gamma \mathbf{P}\mathbf{v}_i - (\mathbf{r} + \gamma \mathbf{P}\mathbf{v}_j)\,|| = \gamma ||\mathbf{P}(\mathbf{v}_i - \mathbf{v}_j)|| \\ &\leq \gamma ||\mathbf{P}||\; ||\mathbf{v}_i - \mathbf{v}_j|| = \gamma ||\mathbf{v}_i - \mathbf{v}_j|| < ||\mathbf{v}_i - \mathbf{v}_j|| \end{aligned} \tag{20.12}$$

We can also prove that $B(\mathbf{v})$ is a contraction mapping by showing that the norm of its Jacobian matrix is smaller than 1. We first find the Jacobian matrix of $B(\mathbf{v})$, its derivative with respect to its vector argument $\mathbf{v}$:

$$\mathbf{J}_B = \frac{d}{d\mathbf{v}} B(\mathbf{v}) = \frac{d}{d\mathbf{v}} (\mathbf{r} + \gamma \mathbf{B}\mathbf{v}) = \gamma \mathbf{P} \tag{20.13}$$

We further find its p-norm with $p = \infty$ (equivalent to $p = 1,\ 2$), the maximum absolute row sum:

$$||\mathbf{J}_B||_{p=\infty} = ||\gamma \mathbf{P}||_\infty = \gamma ||\mathbf{P}||_\infty = \gamma \max_{1 \leq i \leq N} \sum_{j=1}^{N} |P_{ij}| = \gamma < 1 \tag{20.14}$$

The last equality is due to Eq. (20.3), i.e., $||\mathbf{P}||_\infty = 1$.

Having shown that $B(\mathbf{v})$ is a contraction mapping, we can now use the method

of fixed-point iteration first introduced in Section 1.3 to solve the Bellman equation by iterating Eq. (20.11) from an arbitrary initial value, such as $\mathbf{v}_0 = \mathbf{0}$:

$$\mathbf{v}_{n+1} = B(\mathbf{v}_n) = \mathbf{r} + \gamma \mathbf{P}\mathbf{v}_n \tag{20.15}$$

This iteration will always converge to the root of the equation, the fixed point of function $B(\mathbf{v})$.

In summary, the Bellman equation in Eq. (20.8) can be solved by either of the two methods in Eqs. (20.10) and (20.15), so long as $\gamma < 1$. When the size of state space, the cardinality $|S| = N$ of the set of all states, is large, the complexity $O(N^3)$ is high enough that the iterative method may be more efficient.

20.1.3 Markov Decision Process

A *Markov decision process (MDP)* is an extension of the MRP that also includes a certain decision-making rule called *policy* denoted by π. Following a given policy the agent takes an action $a \in A(s)$ out of all actions available at a state s, denoted by $A(s)$, to control the state transition of the dynamic process in the system. An MDP can therefore be described by a quintuple (5-tuple) $< S, P, R, A, \gamma >$, where A represents the set of all actions. A policy can be either deterministic denoted by $a = \pi(s)$, or stochastic denoted by $\pi(a|s) = P(a_t = a|s_t = s)$ as the conditional probability of taking action a in state s. A policy is *soft* if any of the actions available at a state is possible to be taken, i.e., $\pi(a|s) > 0$ for all $a \in A(s)$ and all $s \in S$. In particular, a policy is ϵ-soft if $\pi(a|s) \geq \epsilon/|A(s)|$ for some small value of ϵ.

As there are $|A(s^n)|$ available actions to take in each state $s^n \in S$, there are in total $|A(s^1)| \cdots |A(s^N)| = \Pi_{n=1}^{N} |A(s^n)|$ possible deterministic policies. If $A(s) = A$ is the same for all $|S|$ states, then the total number of policies is $|A|^{|S|}$. Following a given policy from an initial state s_0 will result in a sequence of state transitions called a *trajectory* of the episode:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots \xrightarrow{a_{T-1}} s_T \tag{20.16}$$

of which the state s_t visited at the t-th step can be any of the N states in S. Note that in an episode some of the states in S may be visited multiple times, while some others may never be visited at all. Also note that due to the random nature of the MDP, following the same policy may not result in the same trajectory.

Along the trajectory of an episode starting from state s_0, the discounted rewards from all states visited will accumulate, represented by the value $v(s_0)$, the expectation of the sum of all discounted future rewards. The goal of planning is to find the optimal policy as a sequence of actions for a given MDP model of the environment, so that the $v(s_0)$ is maximized. This optimization problem can be solved by the method of dynamic programming.

In an MDP, both the next state s' and reward r are random variables with joint probability $P(s', r|s, a) = P(s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a)$ conditioned on the action a and the previous state s. The state transition and reward

probabilities can be found by marginalization (similar to Eq. (20.5) for an MRP):

$$P(s'|s,a) = \sum_{r \in R} p(s', r|s, a)$$
$$P(r|s,a) = \sum_{s' \in S} P(s', r|s, a) \tag{20.17}$$

and the expected reward r received after arriving at the current state s is

$$r(s,a) = E[r_{t+1}|s_t = s, a_t = a] = \sum_r r \; P(r|s,a) \tag{20.18}$$

Following a given random policy $\pi(a|s)$, the agent takes an action $a \in A(s)$ to transit from the current state $s_t = s$ to the next state $s_{t+1} = s'$ with probability

$$P_\pi(s'|s) = P(s_{t+1} = s'|s_t = s) = \sum_{a \in A(s)} \pi(a|s) \; P(s'|s,a) \tag{20.19}$$

satisfying $\sum_{s'} P_\pi(s'|s) = 1$, and receives a reward in state s:

$$r_\pi(s) = E_\pi[r_{t+1}|s_t = s] = \sum_{a \in A(s)} \pi(a|s) \; r(s,a) \tag{20.20}$$

where $E_\pi[x]$ denotes the expectation (of some random variable x inside the brackets) with respect to a certain policy $\pi(a|s)$. The summation over all available actions $a \in A(s)$ in state s will be abbreviated to a in the following.

Consider, as a simple and typical example, the *gridworld* problem, where the environment is a rectangular 2-D space divided into squares, each considered as a state with certain reward. The goal for the agent in this problem is to start at a specified position (e.g., the square at the upper-left corner) and travel to a destination (e.g., the square at the lower-right corner) while trying to accumulate the maximum amount of reward along the way. There are four actions (going up, right, down, and left) available at an internal square, but only three or two at a square on the boarder or in the corner of the grid. Upon arrival at a square as a state s, the agent receives a reward r, and then makes a state transition to one of the neighboring squares as the next state s', depending on the action a it takes while following a certain policy π. Note that both the reward r and the next state s' can be either deterministic or probabilistic, and so is the policy π.

Moreover, certain constraints may be imposed on the problem. For example, some of the squares are not allowed to enter, such as the walls in a maze. In this case the problem becomes more challenging, not only because the available actions at some states are reduced, but also some paths may lead to a dead end if the policy is to near-sighted (not far-sighted enough).

In reinforcement learning, to prepare for the algorithms, we further define two important functions with respect to a given policy π of an MDP:

- The *state value function* $v_\pi(s)$ is the expected return in state s of the MDP

while following policy π:

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|s_t = s] \\ &= \sum_a \pi(a|s) \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a)\ v_\pi(s') \right) \\ &= \sum_a \pi(a|s)\ q_\pi(s,a) \end{aligned} \tag{20.21}$$

This is similar to the value function $v(s)$ of an MRP in Eq. (20.8), but now treated as a function of action a as well as state s, based on $q_\pi(s,a)$ as defined below.

- The *state-action value function* $q_\pi(s,a)$ is the expected return of taking a specific action a (regardless of policy π) in state s, and then following π in all subsequent states:

$$\begin{aligned} q_\pi(s,a) &= E_\pi\left[G_t|s_t = s, a_t = a\right] \\ &= r(s,a) + \gamma \sum_{s'} P(s'|s,a)\ v_\pi(s') \\ &= r(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')\ q_\pi(s',a')) \end{aligned} \tag{20.22}$$

where $v_\pi(s')$ is recursively represented as a weighted sum of $q_\pi(s',a')$ based on Eq. (20.21).

The state-action value $q_\pi(s,a)$ plays a more important role than $v_\pi(s)$ as it allows the freedom of taking any action independent of a given policy π, thereby allowing the opportunity to improve an existing policy, e.g., by taking a *greedy* action to maximize the value $q_\pi(s,a)$. The state-action value function $q_\pi(s,a)$ is often abbreviated as the action value or simply *Q-value* for convenience in future discussions.

As a function of the state-action pair (s,a), the state-action value $q_\pi(s,a)$ can be represented as a table of $|S|$ rows, each for one of the states s, and $|A|$ columns each for one of the actions a. The Q-value for each state-action pair stored in the table can be updated iteratively by various algorithms for learning the Q-values.

When in particular the policy is deterministic with $\pi(a|s) = 1$ and $a = \pi(s)$, then we have

$$P_\pi(s'|s) = P(s'|s,a), \qquad r_\pi(s) = r(s,a) \tag{20.23}$$

and the Bellman equations in Eqs. (20.21) and (20.22) become the same:

$$v_\pi(s) = r(s,a) + \gamma \sum_{s'} P(s'|s,a)\ v_\pi(s') = q_\pi(s,\pi(s)) \tag{20.24}$$

This is the same as the Bellman equation of an MRP in Eq. (20.8), which can be solved iteratively if $\gamma < 1$, just as in Eq. (20.15).

Fig. 20.2 illustrates the Bellman equations in Eqs. (20.21) and (20.22), showing the bootstrapping of the state value function in the dashed box on the left, and

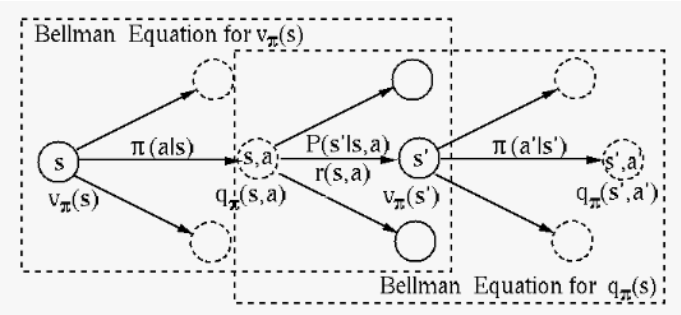


Figure 20.2 Bellman Equations for State and Action Values

that of the action value function in dashed box on the right. Here the word bootstrapping means the iterative method that updates the estimated value at a state s based on the previously estimated value of the next state s'. After taking one of the actions $a \in A(s)$ based on policy $\pi(a|s)$, an immediate reward $r(s,a)$ is received.

Given a policy, it is still not certain which state the MDP will transit into as the next state s' due to the random nature of the state transition described by $P(s'|s,a)$. This uncertainty is represented symbolically by a dashed circle called a *chance node* associated with the action value $q_\pi(s,a)$ as the sum of the immediate reward $r(s,a)$ and the expected value of the next state, the weighted average of values $v(s')$ of all possible state $s' \in S$.

20.2 Model-Based Planning

Given a complete MDP model in terms of $p(s', r|s, a)$ of the environment, the goal of model-based planning is to find the optimal policy π^* that achieves the maximum value:

$$v^*(s) = \max_\pi v_\pi(s), \qquad \pi^*(s) = \arg\max_\pi \ v_\pi(s) \geq \pi(s) \qquad \forall \pi \tag{20.25}$$

based on the partial order that compares different policies according to their corresponding values:

$$\begin{aligned} \text{If} \quad & v_{\pi_1}(s) \geq v_{\pi_2}(s) \geq \cdots v_{\pi_k}(s), \quad \forall s \in S \\ \text{Then} \quad & \pi_1 \geq \pi_2 \geq \cdots \geq \pi_k \end{aligned} \tag{20.26}$$

The problem of policy optimization can be addressed based on two subproblems: *policy evaluation* to measure how good a policy is, and *policy improvement* to get a better policy with higher values.

One way to solve this optimization problem is to use brute-force search to enumerate all $|A(s)|$ available policies at each of the $|S|$ states $s \in S$. If we assume the numbers of available actions in all states are the same for simplicity, i.e.,

$A(s^1) = \cdots = A(s^N) = A$, then the computational complexity of this method is $O(|A|^{|S|})$, likely to be too high for the brute-force search to be practical if $|A|$ is large.

A more efficient and therefore more practical way to optimize the policy is to do it iteratively. Consider a simpler task of improving upon an existing deterministic policy π at a single transition step from the current state s to a next state s'. Instead of taking the action $a = \pi(s)$ dictated by the policy, we can improve it by taking the greedy action that maximizes the action value function in Eq. (20.22), while subsequent steps still follow the old policy π. Now we get a new policy:

$$\pi'(s) = \arg\max_a q_\pi(s,a) = \arg\max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_\pi(s') \right] \tag{20.27}$$

so that the action value is higher than that of the old one:

$$\begin{aligned} q_\pi(s,\pi'(s)) &= \max_a q_\pi(s,a) = \max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_\pi(s') \right] \\ &\geq q_\pi(s,\pi(s)) = v_\pi(s) \end{aligned} \tag{20.28}$$

where the last equality is due to Eq. (20.24) for a deterministic policy.

While it is obvious that this greedy method will result in a higher return for the single state transition from s to s', we still need to prove that by following this greedy method in all subsequent states, we will get a new policy π' that achieves higher values than those of the old one π at *all* states, i.e.,

$$v_{\pi'}(s) \geq v_\pi(s), \qquad \forall s \in S \tag{20.29}$$

This is the *policy improvement theorem*, which can be proven by recursively applying the greedy method to replace the old policy $\pi(s)$ by the new one $\pi'(s)$ for a higher value for each of the subsequent steps one at a time:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s,\pi'(s)) = \max_a q_\pi(s,a) = \max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_\pi(s') \right] \\ &= r(s,\pi'(s)) + \gamma \sum_{s'} P(s'|s,\pi'(s)) v_\pi(s') \\ &\leq r(s,\pi'(s)) + \gamma \sum_{s'} P(s'|s,\pi'(s)) \max_{a'} q_\pi(s',a') \\ &= r(s,\pi'(s)) + \gamma \sum_{s'} P(s'|s,\pi'(s)) \left[r(s',\pi'(s)) + \gamma \sum_{s''} P(s''|s',\pi'(s')) v_\pi(s'') \right] \\ &= r(s,\pi'(s)) + \gamma r(s',\pi'(s)) + \gamma \sum_{s'} P(s'|s,\pi'(s)) \left[\gamma \sum_{s''} P(s''|s',\pi'(s')) v_\pi(s'') \right] \\ &\leq \cdots \\ &= r(s,\pi'(s)) + \gamma r(s',\pi'(s')) + \gamma^2 r(s'',\pi'(s'')) + \cdots \\ &= v_{\pi'}(s) \end{aligned} \tag{20.30}$$

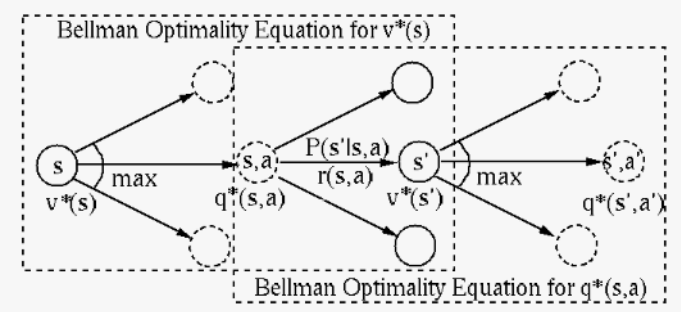


Figure 20.3 Bellman Optimality Equations for State and Action Values

i.e., $\pi'(s) \geq \pi(s)$.

(This derivation may seem long and difficult to follow (although each step should be straightforward), a less experienced reader may choose to skip it and move on when studying the topic the first time.)

The optimal state value $v^*(s)$ and action value $q^*(s,a)$ can therefore be achieved by replacing the weighted average over all possible actions at each state (s and s') in the Bellman equations in Eqs. (20.21) and (20.22) by their maximum:

$$v^*(s) = \max_a q^*(s,a) = \max_a \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) v^*(s') \right) \tag{20.31}$$

and

$$q^*(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) v^*(s') = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} q^*(s',a') \tag{20.32}$$

If the state transition is deterministic, this equation becomes

$$q^*(s,a) = r(s,a) + \gamma \max_{a'} q^*(s',a') \tag{20.33}$$

Similar to Eqs. (20.21) and (20.22) illustrated in Fig. 20.2, Eqs. (20.31) and (20.32), called *Bellman optimality equations*, are illustrated in Fig. 20.3.

The Bellman optimality equations should hold for all N states in $S = \{s^1, \cdots, s^N\}$ and can therefore be written in vector form:

$$\mathbf{v}^* = \max_a (\mathbf{r}_a + \gamma \mathbf{P}_a \mathbf{v}^*) \tag{20.34}$$

where

$$\mathbf{v}^* = \begin{bmatrix} v^*(s^1) \\ \vdots \\ v^*(s^N) \end{bmatrix}, \qquad \mathbf{r}_a = \begin{bmatrix} r(s^1, a) \\ \vdots \\ r(s^N, a) \end{bmatrix}$$
$$\mathbf{P}_a = \begin{bmatrix} P(s^1|s^1, a) & \cdots & P(s^N|s^1, a) \\ \vdots & \ddots & \vdots \\ P(s^1|s^N, a) & \cdots & P(s^N|s^N, a) \end{bmatrix} \tag{20.35}$$

This nonlinear equation system can be solved iteratively for $\mathbf{v}^*$ (similar to Eq. (20.15):

$$\mathbf{v}^*_{n+1} = \max_a (\mathbf{r}_a + \gamma \mathbf{P}_a \mathbf{v}^*_n) = B(\mathbf{v}_n) \tag{20.36}$$

This iteration will converge, as function $B(\mathbf{v})$ is a contraction mapping with $\gamma < 1$:

$$\begin{aligned} ||B(\mathbf{v}_i) - B(\mathbf{v}_j)|| &= || \max_{a_i} (\mathbf{r}_{a_i} + \gamma \mathbf{P}_{a_i} \mathbf{v}_i) - \max_{a_j} \left(\mathbf{r}_{a_j} + \gamma \mathbf{P}_{a_j} \mathbf{v}_j\right) || \\ &\leq || \max_{a_i} (\mathbf{r}_{a_i} + \gamma \mathbf{P}_{a_i} \mathbf{v}_i - \mathbf{r}_{a_i} - \gamma \mathbf{P}_{a_i} \mathbf{v}_j) || \\ &= \max_{a_i} \gamma ||\mathbf{P}_{a_i} (\mathbf{v}_i - \mathbf{v}_j)|| \leq \max_{a_i} \gamma ||\mathbf{P}_{a_i}|| \, ||\mathbf{v}_i - \mathbf{v}_j|| \\ &= \gamma ||\mathbf{v}_i - \mathbf{v}_j|| < ||\mathbf{v}_i - \mathbf{v}_j|| \end{aligned} \tag{20.37}$$

where $||\mathbf{P}|| = ||\mathbf{P}||_\infty = 1$ is the p-norm ($p = \infty$) of a stochastic matrix, which is known to be 1.

This iteration will be used for policy evaluation in the algorithm *value iteration* to be considered below for finding the optimal policy.

Given the complete information of the MDP model in terms of $p(s', r|s, a)$, we can find the optimal policy π^* using either *policy iteration (PI)* or *value iteration (VI)* based on the method of dynamic programming.

- **Policy Iteration**

  Policy iteration carries out the following two intertwined and interacting processes iteratively from some arbitrary initial policy π until convergence when π no longer changes:

  – Policy evaluation:

    Given a deterministic policy π, Eq. (20.21) becomes the Bellman equation in Eq. (20.24), a linear equation system of $N = |S|$ equations of N unknowns $v_\pi(s^1), \cdots, v_\pi(s^n)$, which can be solved iteratively as in Eq. (20.15) to find values at all states:

    $v^\pi_{n+1}(s) = r(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) v^\pi_n(s') \qquad \forall s \in S$

  – Policy improvement:

    Given $v_\pi(s)$ based on policy π, get a better policy π' by the greedy method as in Eq. (20.27):

    $\pi'(s) = \arg\max_a [r(s, a) + \gamma \sum_{s'} P(s'|s, \pi(s)) v_\pi(s')] \qquad \forall s \in S$

We see that in each round of the policy iteration, the processes of policy evaluation and policy improvement, are carried out alternatively, one depending on the other, and one completing before the other starts. The value functions $v_\pi(s)$ at all states are systemically evaluated before they are updated, and the policy $\pi(a|s)$ at all states is updated before it is evaluated. These two processes chase each other as a moving target iteratively and eventually converge to the optimal policy π^* that achieves the maximum value v^*. This approach is said to be *synchronous*, as illustrated here:

$$\pi_o \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} v_{\pi^*} = v^* \tag{20.38}$$

The pseudo code for the algorithm is listed below:

Initialize:

$v(s) \in \mathbb{R},\ v(s) = 0$ if s is terminal node, $\pi(s) \in A(s),\ \ \forall s \in S$;

Set tolerance *tol* (small positive value);

repeat

Policy Evaluation (find $v(s)\ \ \forall s \in S$):

repeat

$\delta = 0$

for $\forall s \in S$:

$v'(s) = r(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) v(s')$

$\delta = \max(\delta, |v(s) - v'(s)|)$

until $\delta < tol$

Policy Improvement (by taking greedy action):

$Done = T$

for each $s \in S$

$\pi'(s) = \arg\max_a [r(s, a) + \gamma \sum_{s'} P(s'|s, \pi(s)) v(s')]$

if $\pi'(s) \neq \pi(s),\ \ Done = F$

end for

until *done*

- **Value Iteration:**

  One drawback of the policy iteration method is the high computational complexity of the inner iteration for policy evaluation. To speed up this process, we can terminate the inner iteration early before convergence. In the extreme case, we simply combine policy evaluation and improvement into a single step, resulting in the following two-step value iteration:

  – Find the optimal values:

  Based on some initial value $v_0(s)$, solve the Bellman optimality equation in Eq. (20.31) by iteration in Eq. (20.36):

$$v_{n+1}(s) = \max_a \left[r(s, a) + \gamma \sum_{s'} P(s'|s, a) v_n(s') \right] \qquad \forall s \in S \tag{20.39}$$

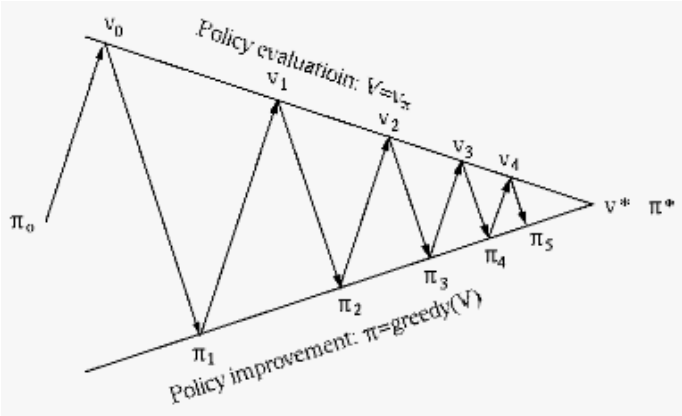


Figure 20.4 Convergence of Policy Iteration

– Find the optimal policy:
Find $\pi^*(s)$ that maximizes $v^*(s)$:

$$\pi^*(s) = \arg\max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_n(s') \right] \qquad \forall s \in S \tag{20.40}$$

Here is the pseudo code for the algorithm:

Initialize: $v(s) = 0 \ \ \forall s \in S$
repeat
$\delta = 0$;
for each $s \in S$
$v'(s) = \max_a [r(s,a) + \gamma \sum_{s'} P(s'|s,a) v(s')]$
$\delta = \max(\delta, |v'(s) - v(s)|)$
$v(s) = v'(s)$
end for
until $\delta < tol$
$\pi^*(s) = \arg\max_a [r(s,a) + \gamma \sum_{s'} P(s'|s,a) v(s')]$

This PI method as illustrated in Fig. (20.4).

20.3 Model-Free Evaluation and Control

The previously discussed dynamic programming methods find the optimal policy based on the assumption that the MDP model of the environment is completely available, i.e., the dynamics of MDP in terms of its state transition and reward mechanism are known. A given policy π can then be evaluated based on the transition probabilities $p(s'|s,a)$, and improved based on greedy action selection

at each state. This model-based planning is an optimization problem involving no learning of the environment.

Here we consider the optimization problem with an unknown MDP model of the environment. In this model-free problem, called a *control* problem, both the state transition and reward probabilities are unknown, and the value functions can no longer be calculated directly based on Eqs. (20.21) and (20.22) as in model-based planning. Now the dynamics of the system in terms of its MDP model needs to be learned based on sampling the environment by repeatedly running a large number of episodes of the stochastic process of the environment while following some given policy, and estimate the value functions as the average of the actual returns received during the sampling process. Such a method is generally referred to as the *Monte Carlo (MC) method*. At the same time, the given policy can also be improved to gradually approach optimality. This approach is called *general policy iteration (GPI)*, as illustrated below, similar to the PI method for model-based planning illustrated in Eq. (20.38).

$$\pi_o \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} q_{\pi^*} = v^* \tag{20.41}$$

While GPI in Eq. (20.41) and GI in Eq. (20.38) may look similar to each other, they are essentially different in the following aspects.

- Policy improvement is based on the state value function $v_\pi(s)$ in model-based planning, but the action-value function $q_\pi(s, a)$ in model-free control. This is because without a specific MDP model the state value function can no longer be calculated, while the action value function can still be estimated based on the actual rewards received while sampling the environment. Similar to how we improve the policy by taking the greedy action to achieve a higher state value in model-based planning, here we improve the policy by taking an action different from that dictated by the given policy π to achieve a higher action value based on some ϵ-*greedy method* to be discussed later.
- Estimating the action value by the MC method requires running a large number of episodes while sampling the environment, and the computational complexity for policy evaluation in the model-free case is much higher than that in model-based planning. To speed up this process in model-free control, the iterative policy evaluation is no longer fully carried for it to converge to the true action value of the policy. Instead, the iterative update of the action value is carried out only once, followed immediately by the next phase of policy improvement. Such an estimated action value is inevitably very noisy, but due to the much reduced complexity, we can afford to run a large number of episodes while sampling the environment.
- In model-based planning, the action value is found based on all available actions at each state (Eq. (20.22)) during policy evaluation, and we only need to *exploit* the greedy action at each state to find the policy that achieves

the maximum action value $q_\pi(s)$ (Eq. (20.27)) during policy improvement. However, in model-free control, we have to learn the action value $q_\pi(s, a)$, as a function of action $a \in A(s)$ and state s by sampling the environment. This estimated action value based only on some partial sample data is inevitably noisy, especially in the early stage of learning when many of the states have not been visited yet. We therefore need to *explore* all actions available at each state to better learn the value function, as well as to exploit the greedy action to improve the policy.

As indicated by the last point above, a proper tradeoff between the *exploitation* of the greedy action and the *exploration* of other non-greedy actions needs be made, so that the agent can be exposed to all possible state-action pairs and gradually learn the action value function while at the same time it can also improve the policy being followed. The *ϵ-greedy method* is just such a method. Specifically, we define $\epsilon \in [0,\ 1]$ as the probability of exploring all actions $A(S)$ available in state s (including the greedy one), each with an equal probability $\epsilon/|A(s)|$, and $1-\epsilon$ as the probability of taking only the greedy action. Therefore the overall probability of taking the greedy action, also one of the $|A(s)|$ available actions, is $1 - \epsilon + \epsilon/|A(s)|$. The policy can now be expressed as

$$\pi'(a|s) = \begin{cases} \arg\max_a q(s,a) & P_{exploitation} = 1 - \epsilon + \epsilon/|A(s)| \\ \text{any other } a \in A(s) & P_{exploration} = \epsilon/|A(s)| \end{cases} \tag{20.42}$$

For example, if $|A(s)| = 4$ and $\epsilon = 2/3$, then the probability of taking the greedy action is $1 - \epsilon + \epsilon/|A(s)| = 3/6 = 1/2$, and the probability of taking any of the remaining three non-greedy actions is $\epsilon/|A(s)| = 1/6$. Such a policy is different from the original policy currently followed by the agent. Such a policy is called *behavior policy*, as it describes the behavior of the agent.

The value of ϵ may vary during the process. A larger ϵ (close to 1) can be used to allow more state-action pairs to be explored at the early stage when many of them have not been visited yet, while smaller ϵ (approaching 0) may be used to exploit the greedy actions for higher values in the later stage when the action value function has been more reliably learned. For example, we can let $\epsilon_k = 1/k$ with k being the number of iterations completed so far, so that it's value gets progressively smaller from a large initial value. As k approaches infinity and ϵ approaches zero, the ϵ-greedy method approaches absolute greedy method towards the end of the process.

Recall we proved the policy improvement theorem in Eq. (20.30) stating that the state value $v_{\pi'}(s)$ achieved by the greedy policy $\pi'(s)$ is no lower than $v_\pi(s)$ achieved by the original policy π, i.e., $\pi'(s) \geq \pi(s)$. Similarly here we can also prove the same policy improvement theorem stating that the action value $q_{\pi'}(s, a)$ achieved by the ϵ-greedy policy π' is no lower than $q_\pi(s, a)$ achieved by

the original policy π:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\ &= \frac{\epsilon}{|A(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \max_a q_\pi(s,a) \\ &\geq \frac{\epsilon}{|A(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \sum_a \frac{\pi(a|s) - \epsilon/|A(s)|}{1-\epsilon} q_\pi(s,a) \end{aligned} \tag{20.43}$$

where the inequality is due to the fact that the maximum value of $q_\pi(s,a)$ is no less than the average of action values $q_\pi(s,a)$ over all $a \in A(s)$ weighted by some normalized coefficients adding up to 1:

$$\sum_a \frac{\pi(a|s) - \epsilon/|A(s)|}{1-\epsilon} = \frac{1}{1-\epsilon} \sum_a \left(\pi(a|s) - \frac{\epsilon}{|A(s)|} \right) = \frac{1}{1-\epsilon}(1-\epsilon) = 1 \tag{20.44}$$

Continuing the equation above we further get

$$\begin{aligned} q_\pi(s, \pi'(s)) &\geq \frac{\epsilon}{|A(s)|} \sum_a q_\pi(s,a) - \frac{\epsilon}{|A(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a|s) q_\pi(s,a) \\ &= \sum_a \pi(a|s) q_\pi(s,a) = v_\pi(s) \end{aligned} \tag{20.45}$$

The iterative process of the GPI method for model-free control is illustrated in Fig. 20.5, which is similar to the PI method for model-based planning based planning illustrated in Fig. 20.4, but here the state value $v_\pi(s)$ is replaced by the action value $q_\pi(s,a)$, and the greedy method is replaced by the ϵ-greedy method.

Comparing the methods of PI and GPI, we note that PI is a *synchronous* method in the sense that the value functions at all states are evaluated before the policy is updated, while GPI is an *asynchronous* method, in which the policy evaluation and improvement are carried out in-place for only one state at a time before all other states, and the two alternating processes of policy evaluation and policy improvement are carried out iteratively.

Also, in GPI the estimated action value $\tilde{q}$ is obtained with only one iteration of policy evaluation, instead of a more accurate estimate that would otherwise be obtained if the iterative evaluation were fully carried out to its convergence, represented by the top line in Fig. 20.5. At the same the time, the ϵ-greedy method is carried out based on the estimated $\tilde{q}$ to improvement the policy, represented by the bottom line in the figure.

The dynamic process of the environment is sampled by running a large number of episodes (all assumed to terminate) of the process while taking action $a_t = \pi(s_t)$ in each state s_t, $(t = 1, \cdots, T)$:

$$s_0 \to (a_1, r_1, s_1) \to (a_2, r_2, s_2) \to \cdots \to (a_T, r_T, s_T) \tag{20.46}$$

and estimating the value functions based on the averaged returns G_t actually received from these episodes. The trajectory of each episode is also illustrated in Fig. 20.6.

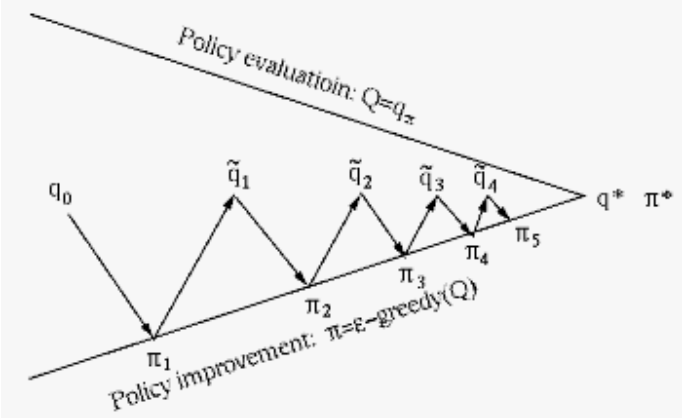


Figure 20.5 Convergence of General Policy Iteration

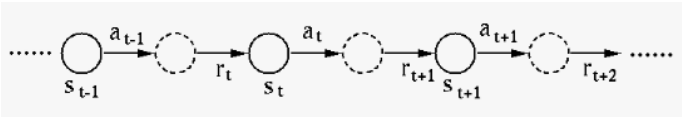


Figure 20.6 A Trajectory Segment of GPI

The trajectories of different episodes are different from each other in general due to the random nature of the environment. In the following sections we will consider specific algorithms for the implementation of the general model-free control discussed above.

To prepare for the discussion of some specific GPI methods, we first consider a general problem of estimating the value of a random variable x as the running average $\hat{x}_n$ of a set of n samples $x_1, \cdots, x_n$ collected in real time, which is updated incrementally upon receiving a new sample x_{n+1}:

$$\begin{aligned}\hat{x}_{n+1} &= \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{1}{n+1}\left(x_{n+1} + \sum_{i=1}^{n} x_i\right) \\ &= \frac{1}{n+1}(x_{n+1} + n\hat{x}_n) = \hat{x}_n + \frac{1}{n+1}(x_{n+1} - \hat{x}_n) \\ &= \hat{x}_n + \alpha(x_{n+1} - \hat{x}_n) \end{aligned} \tag{20.47}$$

where $\hat{x}_n$ and $\hat{x}_{n+1}$ are respectively the old and new estimates of variable x, and $\alpha = 1/(n+1)$ is the step size. If we generalize α for it to take any value between 0 and 1, then Eq. (20.47) above can be rewritten as below for three special cases

for $\alpha = 0,\ 1/(n+1),\ 1$:

$$\begin{aligned}\hat{x}_{n+1} &= \hat{x}_n + \alpha(x_{n+1} - \hat{x}_n) \\ &= \begin{cases} x_{n+1} & \alpha = 1 & \text{no contribution from previous samples} \\ \frac{1}{n+1}\sum_{i=1}^{n+1} x_i & \alpha = 1/(n+1) & \text{equal contribution from all samples} \\ \hat{x}_n & \alpha = 0 & \text{no contribution from latest sample} \end{cases}\end{aligned} \tag{20.48}$$

We see that by adjusting α, we can control how much the latest and previous samples contribute to the to estimated average. A reasonable strategy is therefore to progressively decrease α from its initial value 1 down to 0, so that the estimated average is mostly determined by the recent samples in the early stage of the process but it becomes progressively more stabilized when enough samples have been collected in the later stage. Such a strategy is suitable in general in estimating model parameters of a non-stationary system, such as an MDP with varying behaviors when the policy is being modified continuously.

The updating process in Eq. (20.47) can be considered as one iterative step in the method of stochastic gradient descent (SGD) (Section 6.1) for minimizing the squared error $\varepsilon = (x_{n+1} - \hat{x}_n)^2/2$ between the old average $\hat{x}_n$ and the latest sample x_{n+1}, and the second term is the gradient $d\varepsilon/d\hat{x}_n$ weighted by the step size.

In the specific case of model-free control, Eq. (20.47) is used to iteratively update the estimated state value functions while sampling the environment. As we will see in the following sections, the state-action values are estimated as the average of all previous sample returns, and they are updated incrementally when a new sample return G_{n+1}, the *target*, becomes available:

$$\begin{aligned} v_{n+1}(s) &\Leftarrow v_n(s) + \alpha(G_n - v_n(s)) \\ q_{n+1}(s,a) &\Leftarrow q_n(s,q) + \alpha(G_n - q_n(s,a)) \end{aligned} \tag{20.49}$$

The result of this iterative learning process is the estimated state-action values $q(s,a)$, stored in a 2-D array, called a *Q-table*, of which the rows are for all the states and the columns are for all available actions at each state. Based on the Q-table, the optimal control policy can be determined by always taking the greedy action with the maximum state-action value at each state, and the solution of the reinforcement learning problem can be found by following the sequence of optimal actions as in shown Eq. (20.16) from the given initial state s_0. Such methods based on the Q-table are called the *tabular methods*.

20.3.1 Monte Carlo (MC) Algorithms

We first consider the simple problem of evaluating an existing policy π in terms of its value function given by the Bellman equation in Eq. (20.24):

$$v_\pi(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) v_\pi(s') \tag{20.50}$$

This equation for the model-based case can also be used in the model-free case based on the MC method by running multiple sample episodes in the environment. The state value $v_\pi(s)$ is estimated as the average of the actual returns G_t, which in turn are calculated as the sum of discounted rewards $r_{t+1}, \cdots, r_T$ from the current state $s = s_t$ onward to the terminal state s_T found at the end of the episode (assuming a finite horizon).

This method has two versions:

- *Every-visit*: Returns of all visits to a state in the trajectory of an episode are counted in the estimation of the state value function.
- *First-visit*: only the first visit to each state in each episode is counted.

In the first method, all sample returns G_t from multiple visits to a state s_t contribute to the estimated state value $v_\pi(s_t)$. But these sample returns for the state may not be independent of each other as the later ones are dependent on the previous ones, i.e., they are not i.i.d. samples and the estimate values may be biased. This problem is avoided in the second method, where only the return of the first visit to each of the states is used as an independent sample drawn from the distribution, and the estimated values are no longer biased. However, as only a fraction of all samples in the trajectory is used to estimate the state values, the cost of the unbiased estimation is its high variance, which can be reduced only by running a large number of episodes to get enough samples for a more statistically reliable average. We realize that this is just the common issue of bias-variance tradeoff (Section 4.2) faced by many learning algorithms.

The pseudo code below is for first-visit version due to the if statement, which can be removed for the every-visit version.

Input: a given policy π to be evaluated
Initialize: $G_t = [\ ]$ (empty lists) for all $t = 0, \cdots, T-1$
loop (for each episode)
    run episode to the end while following π to get sample data:
        $(s_0, a_0, r_1),\ (s_1, a_1, r_2), \cdots, (s_{T-1}, a_{T-1}, r_T)$
    $G = 0$
    for $t = T-1, T-2, \cdots, 0$ (for each step)
        $G = \gamma G + r_{t+1}$
        if $S_t \notin \{S_0, \cdots, S_{t-1}\}$ (first visit)
            $G_t = [G_t, G]$ (append G to list G_t)
$V(S_t) = average(G_t),\ \forall t = 0, \ , \cdots, T-1$

In this code, the function *average* finds the average of all elements of a list when all sample returns are available at the end of each episode. A more computationally efficient method (in terms of both space and complexity) is to find the average incrementally as in Eq. (20.49). Based on this incremental average, policy evaluation in terms of both the state and action values can also be carried out by the code below:

Initialize $v(s) = 0,\ \forall s$
loop (for each episode)
    run episode following π to get $G_t,\ \forall t$
    for $t = 0, \cdots, T-1$ (for each step)
        if s_t is visited the first time
            $v(s_t) = v(s_t) + \alpha(G_t - v(s_t))$

and

Initialize: $q(s, a) = 0,\ \forall (s, a)$
loop (for each episode)
    run episode following π to get $G_t,\ \forall t$
    for all (s, a) pairs visited
        if (s, a) is visited the first time
            $q(s, a) = q(s, a) + \alpha(G_t - q(s, a))$

The *if* condition can be removed for every-visit version of the algorithm.

We next consider the MC algorithm for model-free control, based on the generalized policy iteration of alternating and interacting policy evaluation and policy improvement. While sampling the environment by following an existing policy π the action value $q_\pi(s)$, instead of the state value $v_\pi(s)$, is gradually estimated and the policy is gradually improved at the same time. This iteration will converge to the optimal policy at the limit when the number of iterations goes to infinity.

The pseudo code for this algorithm is listed below.

Input: $\epsilon > 0,\ \alpha > 0$, and a policy π
Initialize: $q(s, a) = 0,\ \forall (s, a),\ \pi = \epsilon - \text{soft}$ (arbitrarily)
loop (for each episode)
    run episode following π to get $G_0, \cdots, G_{T-1}$
    for $t = 0, \cdots, T-1$
        if (s_t, a_t) is visited the first time
            $q(s_t, a_t) = q(s_t, a_t) + \alpha(G_t - q(s_t, a_t))$
            $a^* = \arg\max_a q(s_t, a_t)$
            for $\forall a \in A(s_t)$

$$\pi(a|s_t) = \begin{cases} 1 - \epsilon + \epsilon/|A(s_t)| & \text{if } a = a^* \\ \epsilon/|A(s_t)| & \text{else} \end{cases}$$

Here the ϵ-greedy policy $\pi(a|s)$ is based on the action value function $q(s, a)$ (maximized by the greedy action). Through out the iteration, the policy is modified following the action value function, while at the same time the action value function is modified based on the policy, until the process converges to the desired optimality.

The optimal policy $\pi(s)$ obtained by the algorithm above is not completely deterministic due to the ϵ-greedy approach, but it is said to be *near-deterministic* as the greedy action is favored over other non-greedy actions.

20.3.2 Temporal Difference (TD) Algorithms

The temporal difference (TD) method is a combination of the MC method considered above and the bootstrapping DP method based on the Bellman equation. The main difference between the TD and MC methods is the target, the return G, in the incremental average in Eq. (20.49) for estimating either the state or action value functions. While in the MC method the target is the actual return G_t calculated at the end of the episode when all subsequent rewards $r_{t+1}, \cdots, r_T$ are available, here in the TD method the target, called the *TD target*, is the sum of the immediate reward r_{t+1} and the previously estimated value $v_\pi(s_{t+1})$ at the next state:

$$G_t = r_{t+1} + \gamma v_\pi\ (s_{t+1}) \tag{20.51}$$

i.e., the TD method is an in-place bootstrapping method. It makes more efficient use of the sample data, and it updates the estimates of the value functions and policy more frequently at every step of an episode, instead of at the end of the episode as in the MC method. Also, it can be used even if the dynamic process of the environment is non-episodic with an infinite horizon.

Again, we first consider the simpler problem of policy evaluation. As in the MC method, we estimate the value function $v_\pi(s)$ as the average of the actual returns G found by running multiple episodes while sampling the environment. Substituting this TD target into the running average in Eq. (20.49) for the value function above, we get

$$\begin{aligned} v_\pi(s) &\Leftarrow v_\pi(s) + \alpha(G - v_\pi(s)) \\ &= v_\pi(s) + \alpha(r + \gamma v_\pi(s') - v_\pi(s)) = v_\pi(s) + \alpha\delta \end{aligned} \tag{20.52}$$

where δ, called the *TD error*, is defined as the difference between the TD target and the previous estimate:

$$\delta = G - v_\pi(s) = r + \gamma v_\pi(s') - v_\pi(s) \tag{20.53}$$

and $v_\pi(s')$ is the current estimate of the value of the next state previously estimated (bootstrapping). The iteration of this TD method converges to the true value $v_\pi(s)$, if the step size is small enough.

Here is the pseudo code for policy evaluation using the TD method, based on parameters γ and α:

Input: a given policy π to be evaluated
Initialize: $v(s)$ for all $s \in S$ arbitrarily, $v(s) = 0$ for terminal state s, $\alpha \in (0,\ 1]$
loop (for each episode)
    $s = s_0$
    while s is not terminal (for each step)
        take action $a = \pi(s)$, get reward r and next state s'
        $v(s) = v(s) + \alpha[r + \gamma v(s') - v(s)]$
        $s = s'$

We further consider the TD method for model-free control, which is similar to the TD method for model-free evaluation considered before. By running many episodes of the environment, the estimated action values $q(s, a)$ for all state-action pairs, stored in a state-action table or Q-table, are iteratively updated as the running average of the return G, the sum of the immediate reward r and the action value $q(s', a')$ of the next state based on an action a', dictated by the policy being followed, such as the ϵ-greedy policy. This iterative process gradually learns the action value functions of the MDP model of the environment, based on which the optimal action for each state can be obtained as the one with the maximum Q-value.

The TD method can be carried out by two algorithms, with the only difference in terms of how the action value is updated.

- *On-policy* algorithm:

  The action value $q(s, a)$ is updated based on action a' at the current state s dictated by the policy being followed:

$$\begin{aligned} q(s, a) &\Leftarrow q(s, a) + \alpha(G - q(s, a)) \\ &= q(s, a) + \alpha(r + \gamma q(s', a') - q(s, a)) \end{aligned} \tag{20.54}$$

- *Off-policy* algorithm:

  The action value $q(s, a)$ is updated based on the greedy action a' at the current state s regard the policy being followed:

$$q(s, a) \Leftarrow q(s, a) + \alpha(r + \gamma \max_{a'} q(s', a') - q(s, a)) \tag{20.55}$$

These two algorithms become the same if the policy being followed is greedy.

- **State-Action-Reward-State-Action (SARSA)**

  This is an on-policy algorithm, which is called *SARSA* as it updates the Q-value based on the current state s and action a, the immediate reward r, the next state s' and action a'. The pseudo code of SARSA algorithm is listed below, where the policy being followed, e.g., ϵ-greedy, is denote by π.

  The pseudo code of SARSA algorithm is listed below.

  Initialize: $q(s, a) = 0$ for all (s, a), $\alpha \in (0,\ 1]$, $\epsilon > 0$

  loop (for each episode)

    $s = s_0$

    get action a at s according to π based on $q(s, a)$

    while s is not terminal (for each step)

      take action a and get reward r and next state s'

      get action a' according to π

      $q(s, a) = q(s, a) + \alpha[r + \gamma q(s', a') - q(s, a)]$

      $s = s',\ a = a'$

A variation of SARSA, the *expected SARSA*, updates the Q-value in state s based on the expected Q-value of the next state s', the weighted average of the Q-values resulting from all possible actions, instead of only one action. Consequently, the estimated Q-values by the expected SARSA have lower variance than SARSA, and a higher learning rate α can be used to speed up the learning process.

$$q(s,a) = q(s,a) + \alpha \left(r + \gamma \sum_a \pi(a|s') q(s',a) - q(s,a) \right) \tag{20.56}$$

- **Q-Learning**

  This is an off-policy algorithm, which updates the Q-value based on the greedy action instead of that dictated by the policy being followed.

  The pseudo code of the Q-learning algorithm is listed below.

  Initialize: $q(s,a) = 0$ for all (s,a), $\alpha \in (0,\ 1]$, $\epsilon > 0$
  loop (for each episode)
  $s = s_0$
  while s is not terminal (for each step)
  take action a at s according to π based on $q(s,a)$
  get reward r and next state s'
  $q(s,a) = q(s,a) + \alpha[r + \gamma \max_{a'} q(s',a') - Q(s,a)]$
  $s = s'$

Here is the comparison of the MC and TD methods:

- The MC method estimates the state value $v_\pi(s_t)$, the expected return, by the true return G_t obtained at the end of the episode, i.e., the estimated value is updated only once at the end of every episode. The TD method estimates $v_\pi(s_t)$ as the sum of the immediate reward and the action value at the next state previously estimated, i.e., it is a bootstrap method, and the estimated value is updated at every step of every episode.
- MC can only learn episodic (terminating) environments with complete episodes; while TD can learn continuing (non-terminating) environments of incomplete episodes.
- MC estimates based on sample returns G_t, and it is unbiased especially in first-visit version. The TD uses the bootstrap approach to find the TD target based on sample data that are not i.i.d. in general, and it is more sensitive to the initial guess of the value functions, so it is more biased.
- MC based on the sample returns G_t is affected by many random events (state transitions, actions, and rewards), and in particular the first-visit version only makes use of the sample data from the first visit of a state. It does not use the available data efficiently, and it has high variance. The TD method is based on only one random variable, the estimated return, and it updates the action value at every step of the episode, it makes more frequent and efficient use of the sample data, it has lower variance.

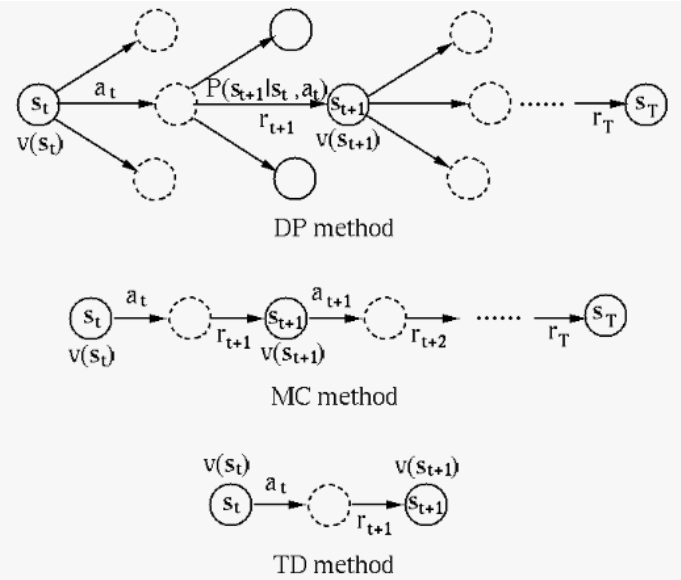


Figure 20.7 Comparison of DP, MC and TD Methods

The dynamic programming (DP) method for model-based planning, and the Monte-Carlo (MC) and time difference (TD) methods for model-free control are summarized in Fig. 20.7.

Example 20.1 The TD method is applied to the grid world problems as shown in Fig. 20.8, where a rectangular array is used to represent a maze of which all positions are sequentially indexed from top left to lower right. This problem is modeled as a Markov decision process with each of the positions treated as a state of the MDP model. For an agent to achieve the goal of finding the shortest path from the starting position (top left corner) to the destination position (lower right corner), we assign a large positive reward (e.g., 10) to the destination position, a large negative reward (e.g., -100) to all positions for the walls of the maze, and a small negative reward (e.g., -1) to all other empty space available for the agent to move into. By maximizing the cumulative reward along its way, the agent will find the desired path.

This maze problem can be solved by the following steps:

- Set up all parameters in the algorithm, including ϵ, α, γ, and the maximum number of iterations for episodes (outer loop) and steps in each of each episode (inner loop).
- Define the maze as a rectangular array with reward for each position (wall or empty space), together with the start and end positions (as the initial and goal states).
- Initialize the Q-table, here an array of the same size as the maze, to zero.

- Carry out either SARSA or Q-learning algorithm to learn the state-action functions by sampling the environment and updating the Q-table iteratively.
- Get the solution, the optimal path from the initial state to the goal state, with maximum cumulative reward, by taking the greedy action with the maximum Q-value in the Q-table for each state along the path.

The Matlab code listed below are the key functions used for solving the grid world problem.

```
function Q=SARSA(Q,s0,st)           % SARSA Learning
    global maze alpha gamma epsilon max_step max_episode
    ie=1;                           % episode counter
    while ie < max_episode          % for all episodes
        is=1;                       % episode counter
        s=s0;                       % initial state
        a=get_action(s,Q);          % action by e-greedy
        while s~=st                 % for each step in episode
            sp=get_next_state(s,a); % next state s'
            r=get_reward(s,a);      % reward
            ap=get_action(sp,Q);    % naxt action by e-greedy
            Q(s,a)=Q(s,a)+alpha*(r+gamma*Q(sp,ap)-Q(s,a));
            is=is+1;
            s=sp;                   % update state
            a=ap;                   % update action
            if is>max_step
                break;
            end
        end
        ie=ie+1;
    end
end

function Q=Q_Learning(Q,s0,st)      % Q-Learning
    global maze alpha gamma epsilon max_step max_episode
    ie=1;                           % episode counter
    while ie<max_episode            % for all episodes
        is=1;                       % step counter
        s=s0;                       % initial state
        while s~=st                 % for each step in episode
            a=get_action(s,Q);      % action by e-greedy policy
            sp=get_next_state(s,a); % next state s'
            r=get_reward(s,a);      % reward
            Q(s,a)=Q(s,a)+alpha*(r+gamma*qmax(Q(sp,:))-Q(s,a));
            s=sp;                   % update state
```

```
            is=is+1;
            if is>max_step
                break;
            end
        end
        ie=ie+1;
    end
end

function r=get_reward(s,a)
                    % get reward given state and action
    global maze alpha gamma
    sp = get_next_state(s,a);
    [row,col] = state2idx(sp,maze);
    r = maze(row,col);
end

function sp = get_next_state(s,a)
    % get next state given current state s and action a.
    global maze alpha gamma
    [~,n]=size(maze);
    switch a
        case 1       % move up
            sp=s-n;
        case 2       % move right
            sp=s+1;
        case 3       % move down
            sp=s+n;
        case 4       % move left
            sp=s-1;
    end
end

function action = get_action(s, Q)
    global maze alpha gamma epsilon
    % get an action in state s by e-greedy policy
    [M N]=size(maze);
    [row,col]=state2idx(s,maze);
    valid_actions=[];       % two-column array: action and Q-value
                            % check all four directions
    if row > 1              % Up
        r=row-1;
        c=col;
        if maze(r,c)~=-100  % if valid
            valid_actions=[valid_actions; 1, Q(s,1)];
```

```
        end
    end
    if row < M              % Down
        r=row+1;
        c=col;
        if maze(r,c)~=-100  % if valid
            valid_actions=[valid_actions; 3, Q(s,3)];
        end
    end
    if col > 1              % Left
        r=row;
        c=col-1;
        if maze(r,c)~=-100  % if valid
            valid_actions=[valid_actions; 4, Q(s,4)];
        end
    end
    if col < N              % Right
        r=row;
        c=col+1;
        if maze(r,c)~=-100  % if valid
            valid_actions=[valid_actions; 2, Q(s,2)];
        end
    end
    if rand<epsilon         % of probability epsilon
        idx=randi(size(valid_actions,1)); % random index (non-greedy)
    else                    % of probability 1-epsilon
        [q idx]=max(valid_actions(:,2));  % index of maximum Q (greedy)
    end
    action=valid_actions(idx,1);  % e-greedy action

end

function [r, c] = state2idx(s,maze)
    % Convert state index s to row and column indices (r c)
    [~,n]=size(maze);
    r=ceil(s/n);
    c=s-n*(r-1);
end

function [path reward]=find_path(s0, st, Q)
% get a path from s0 to st by taking greedy actions based on Q-table
    global maze alpha gamma epsilon
    Q(Q==0)=-inf;               % convert 0 (invalid actions) to -inf
    reward=0;                   % initialize cumulative reward
    path=[s0];                  % initialize path to start position
```

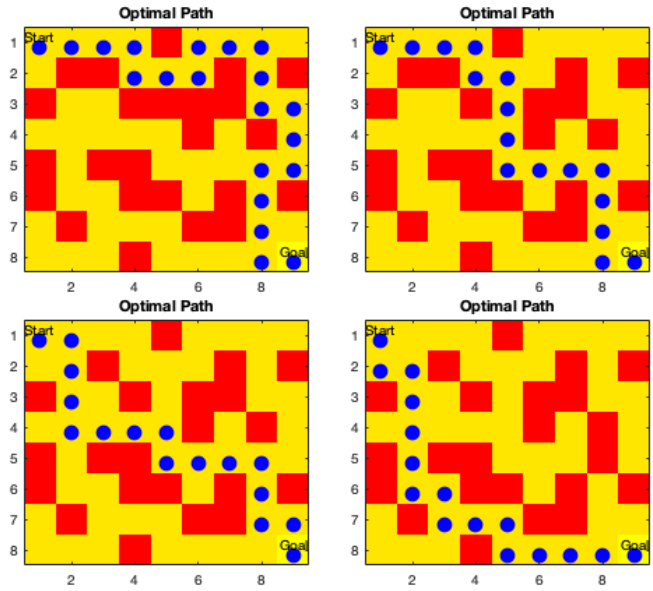


Figure 20.8 Optimal paths in four different mazes

```
    s=s0;
    while path(end)~=st        % add new positions to path
        [v,a]=max(Q(s,:));     % get greedy action a
        reward=reward+get_reward(s,a);   % update cumulative reward
        s=get_next_state(s,a); % get next position
        path=[path,s];         % append new position to path
    end
end
```

The optimal paths found by the code are illustrated in Fig. 20.8 for four slightly different mazes.

20.3.3 TD(λ) Algorithm

The MC and TD methods considered previously can be unified by the n-step TD(λ) method that spans a spectrum, of which the MC and TD methods are two special cases at the opposite extremes.

Recall that both MC and TD algorithms updates iteratively the estimated state-action value function $q(s, a)$ based on Eq. (20.49), but they estimate the target G_t in the equation differently. In an MC algorithm, the target is the actual return $G_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-t-1} r_T$, the sum of discounted rewards of all future steps up to the terminal state s_T, available only at the end of each episode. On the other hand, in a TD algorithm (Eq. (20.51)), the target $G_t = r_{t+1} + \gamma v_\pi(s_{t+1})$ is the sum of the immediate reward r_{t+1} available at each step of

the episode, and the discounted value of the next state, based on bootstrapping. We therefore see that an MC algorithm updates the value functions once every episode, while a TD algorithm updates once every step in the episode.

As a tradeoff between the MC and TD methods, the n-step TD algorithm can be considered a generalization of the TD method, where the target in Eq. (20.49) is an *n-step return*, the sum of the discounted rewards in the n subsequent states and the discounted value function at the following state s_{t+n} with $1 \leq n < \infty$:

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n v_\pi(s_{t+n}) \tag{20.57}$$

In particular, when

- $n = 1$: the 1-step return is the sum of the immediate reward and the estimated value of the next state, the same as the TD target in the TD method:

$$G_{t:t+1} = r_{t+1} + \gamma v(s_{t+1}) \tag{20.58}$$

- $n = T - t$: the n-step return is the sum of the discounted rewards from all future states up to the terminal state at the end of the episode, i.e., it is the return r_t defined in Eq. (20.7), the same as in the MC method:

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T + v_\pi(s_{t+n}) = G_t \tag{20.59}$$

  where the value function of the terminal state $v_\pi(s_{t+n}) = v_\pi(s_T) = 0$ is zero.
- $T - t < n < \infty$: All states s_n beyond the terminal state s_T with $n > T - t$ remain the same as the terminal state with value $v(s_T) = 0$ and return G_t, same as in the MC method.

We see that all these n-step returns form a spectrum with the TD ($n = 1$) and MC ($n \geq T - t$) methods at the two ends.

Based on these n-step returns of different n values, the n-step TD algorithm can be further generalized to a more computationally advantageous and therefore more useful algorithm called TD(λ), by which the MC and TD algorithms are again unified as two special cases at the opposite ends of a spectrum.

We first define the λ-return G_t^λ as the weighted average of all n-step returns for $n = 1, 2, \cdots, \infty$:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{20.60}$$

where $0 \leq \lambda < 1$. Here the coefficient $1 - \lambda$ is needed for the exponentially decaying weights to be normalized:

$$(1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} = (1 - \lambda) \sum_{n=0}^{\infty} \lambda^n = \frac{1 - \lambda}{1 - \lambda} = 1 \tag{20.61}$$

The λ-return G_t^λ defined above can be expressed in two summations: the sum of

the first $T-t-1$ n-step returns $G_{t:t+1}, \cdots, G_{t:T-1}$, and the sum of all subsequent n-step returns $G_{t:T} = \cdots = G_{t:\infty} = G_t$:

$$\begin{aligned} G_t^\lambda &= (1-\lambda)\left[\sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \sum_{n=T-t}^{\infty} \lambda^{n-1} G_t\right] \\ &= (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t, \qquad (0 \le t \le T) \end{aligned} \tag{20.62}$$

where the coefficient of G_t in the second term is the sum of all coefficients in the second summation:

$$(1-\lambda)\sum_{n=T-t}^{\infty} \lambda^{n-1} = (1-\lambda)\sum_{m=0}^{\infty} \lambda^m \lambda^{T-t-1} = \lambda^{T-t-1} \tag{20.63}$$

where $m = n - T + t$. We see that in Eq. (20.62) all n-step returns $G_{t:t+n}$ are weighted, by exponentially decaying coefficient λ^{n-1}, except the true return G_t which is weighted by λ^{T-t-1}.

Again consider two special cases:

- $\lambda = 0$: all terms in Eq. (20.62) are zero, except the first one with $n = 1$ (with $\lambda^0 = 0^0 = \lim_{\lambda \to 0} \lambda^0 = 1$):

$$G_t^{\lambda=0} = \lambda^{n-1} G_{t:t+n}\Big|_{n=1} = 0^0\, G_{t:t+1} = r_{t+1} + \gamma v(s_{t+1}) \tag{20.64}$$

  This is simply the TD target, i.e., TD(0) is the TD method.
- $\lambda = 1$: the first summation is zero and

$$G_t^{\lambda=1} = G_t \tag{20.65}$$

  i.e., TD(1) is the MC method.

We therefore see that TD(λ) is a general algorithm of which the two special cases TD(0) and TD(1) are respectively the TD and MC methods.

In Eq. (20.62), G_t^λ is calculated as the weighted sum of all n-step returns $G_{t:t+n}$ up to the last one G_t available at the end of the episode. This summation can be truncated to include fewer terms before reaching the terminal state at the end of the episode:

$$G_{t:h}^\lambda = (1-\lambda)\sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}, \qquad (0 \le t < h \le T) \tag{20.66}$$

Specially when $h = T$, $G_{t:h}^\lambda = G_t^\lambda$ is the same as in Eq. (20.62), and when $h = t+1$, $G_{t:h}^\lambda = G_{t:t+1} = r_{t+1} + \gamma v(s_{t+1})$ is the TD target.

Based on λ-return, the iterative update of the value function $v_\pi(s_t)$ in Eq. (20.52) can be modified to:

$$v_\pi(s_t) = v_\pi(s_t) + \alpha(G_t^\lambda - v_\pi(s_t)) = v_\pi(s_t) + \alpha\delta_t \tag{20.67}$$

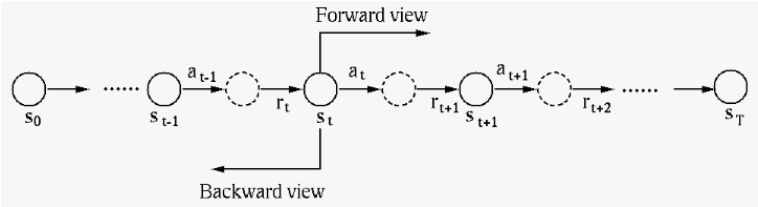


Figure 20.9 Forward View and Backward View

where

$$\delta_t = G_t^\lambda - v_\pi(s_t) \tag{20.68}$$

This iteration is called the *forward view* of TD(λ), which is similar to the MC method, as they are based on either G_t or G_t^λ, available in the future only at the end of each episode, as illustrated in Fig. 20.9.

An alternative but equivalent method is the *backward view* of TD(λ). This method is based the *eligibility trace* $e_t(s)$ of each state, representing the frequency and recency of the visits to each state. If a state s has been more frequently and recently visited compared to others, its eligibility trace $e_t(s)$ is greater than others and its value function $v_\pi(s)$ will be updated by a greater increment than others.

The eligibility trace $e_t(s)$ at all states are initialized to zero. At each step if state $s = s_t$ is currently visited then its eligibility trace is boosted by an increment of 1 to become $e_t(s) = e_t(s) + 1$, while all others (if greater than zero) decay exponentially:

$$e_t(s) = \gamma\lambda e_{t-1}(s) \quad (\forall s \in S) \tag{20.69}$$

The backward view of TD(λ) is similar to the TD(0) method, in that they are both based on the immediate reward r_{t+1} available at each time step of the episode, instead of at the end of the episode, and they are therefore more computationally convenient. Here the iterative update of the estimated value function in Eq. (20.52) is modified so that *all* states (not only the one currently visited) are updated to different extents depending on $e_t(s)$:

$$v_\pi(s) = v_\pi(s) + \alpha\delta_t e_t(s), \qquad \forall s \in S \tag{20.70}$$

where the TD error δ_t is the same as in TD(0), given in Eq. (20.53):

$$\delta_t = r_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t) \tag{20.71}$$

Here is the pseudo code for the backward view of the TD(λ) method for policy evaluation:

Input: policy π to be evaluated
Initialize: $v(s) = 0 \ \ \forall s \in S$
loop (for each episode)

$s = s_0$
$e(s) = 0,\ \forall s \in S$
while s is not terminal (for each step)
take action a based on $\pi(a|s)$, get reward r and next state s'
$\delta = r + \gamma v(s') - v(s)$
$e(s) = e(s) + 1$
for all s
$v(s) = v(s) + \alpha\delta e(s)$
$e(s) = \gamma\lambda e(s)$
$s = s'$

Note that values at all states are updated, but to different extents depending on $e(s)$, different from the TD algorithm where only the value at the state currently visited is updated.

Again consider two special cases:

- $\lambda = 0$, $e(s) = 0$ for all states except the current s being visited, i.e., TD(λ) becomes TD(0), the same as the TD method.
- $\lambda = 1$, $e(s)$ is scaled down by a factor $\gamma < 1$ for all states except the current s being visited, i.e., TD(λ) becomes TD(1), the same as the MC method. However, different from the MC method that updates the value function $v(s)$ at the end of each episode, here $v(s)$ is still updated at every step of the episode due to the backward view of the method.

This backward view of the TD(λ) method can be applied to model-free control when the state value function is replaced by the action value function. Here are two algorithms based on eligibility traces that correspond to the SARSA and Q-learning algorithms based on TD(0).

- SARSA(λ) algorithm:
  Initialize: $q(s,a) = 0,\ \forall(s,a),\ \alpha \in (0,\ 1],\ \epsilon > 0$, denote ϵ-greedy policy $\pi(a|s)$ by π
  loop (for each episode)
  $s = s_0$
  $e(s,a) = 0,\ \forall(s,a)$
  get action a according to π based on $q(s,a)$
  while s is not terminal (for each step)
  $e(s,a) = e(s,a) + 1$
  take action a, get reward r and next state s'
  get action a' according to π based on $q(s',a)$
  $\delta = r + \gamma q(s',a') - q(s,a)$
  for all (s,a)
  $q(s,a) = q(s,a) + \alpha\delta\, e(s,a)$
  $e(s,a) = \gamma\lambda e(s,a)$
  $s = s',\ a = a'$
- Q(λ) algorithm:

Initialize: $q(s,a) = 0,\ \forall(s,a),\ \alpha \in (0,\ 1],\ \epsilon > 0$, denote ϵ-greedy policy $pi(a|s)$ by π
loop (for each episode)
  $s = s_0$
  $e(s,a) = 0,\ \forall(s,a)$
  get action a according to π based on $q(s,a)$
  while s is not terminal (for each step)
    $e(s,a) = e(s,a) + 1$
    take action a, get reward r and next state s'
    get action a' according to π based on $q(s',a)$
    $a^* = \arg\max_b q(s',b)$
    $\delta = r + \gamma q(s',a^*) - q(s,a)$
    for all (s,a)
      $q(s,a) = q(s,a) + \alpha\delta\, e(s,a)$
      if $a' = a^*$ **then** $e(s,a) = \gamma\lambda e(s,a)$ **else** $e(s,a) = 0$
    $s = s',\ a = a'$

Q(λ) algorithm is different from the SARSA(λ) algorithm in two ways. First, the action value is updated based on the greedy action a^*, instead of the a' give by policy π; second, if the ϵ-greedy policy π happens to choose a random non-greedy action $a' \neq a^*$ with probability ϵ to explore rather than exploiting a^*, the eligibility traces of all states are reset to zero. There are other different versions of the algorithm which do not reset these traces.

20.4 Value Function Approximation

All previously considered algorithms are based on either the state value function $v_\pi(s)$ for each state s or the action value function $q_\pi(s,a)$ for each state-action pair. As these function values can be considered as either a 1-D or 2-D table, the algorithms based on such value functions are called *tabular methods*. However, when the number of states and the number of actions in each state are large (e.g., the game Go has 10^{170} states), the tabular methods are not suitable due to the unrealistic table size. In such a case, we can approximate the state value function $v(s)$ or action-value function $q(s,a)$ by a function $\hat{v}(s,\boldsymbol{\theta})$ or $\hat{q}(s,a,\boldsymbol{\theta})$, parameterized by a vector $\boldsymbol{\theta}$, containing a set of parameters of the function as its components. Such a function can be treated as a black box as illustrated in Fig. 20.10. Now the state or action value functions can be approximated based on $\boldsymbol{\theta}$ to be estimated during the learning process when sampling the environment. This approach is more efficient and practical than the tabular methods due to its much smaller number of unknown parameters in comparison to the number of unknown variables in the tabular methods.

Such value function approximators can be implemented by methods such as those listed below:

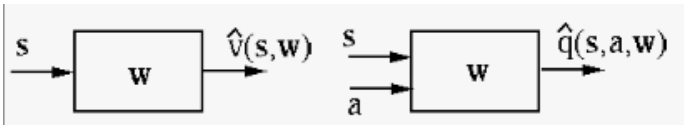


Figure 20.10 Estimation of State and Action Value Functions

- Regression
- Multi-layer neural networks
- Decision trees
- Fourier/wavelet bases

We consider the approach of regression method in this section, and then the neural network method in the next section.

The regression method is to fit a continuous function $\hat{v}(s, \boldsymbol{\theta})$ or $\hat{q}(s, a, \boldsymbol{\theta})$ to a set of discrete and finite data points $v_\pi(s)$ or $q_\pi(s, a)$ obtained by sampling the environment, so that all points in the state space, including those for states or state-action pairs never actually observed during the sampling process, can be estimated by interpolation based on the regression function.

Our task is to find the optimal parameter $\boldsymbol{\theta}$ of the regression model which minimizes the objective function, the mean squared error between the approximated values and the sample values:

$$\varepsilon(\boldsymbol{\theta}) = \frac{1}{2} E_\pi[(v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}))^2] \tag{20.72}$$

The gradient vector of $\varepsilon(\boldsymbol{\theta})$ is

$$\begin{aligned} \mathbf{g}_\varepsilon(\boldsymbol{\theta}) &= \frac{d}{d\boldsymbol{\theta}} \varepsilon(\boldsymbol{\theta}) = \frac{1}{2} E_\pi \left[\frac{d}{d\boldsymbol{\theta}} \left[(v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}))^2 \right] \right] \\ &= -E_\pi \left[(v_\pi(s) - \hat{v}(s, \boldsymbol{\theta})) \frac{d}{d\boldsymbol{\theta}} \hat{v}(s, \boldsymbol{\theta}) \right] \end{aligned} \tag{20.73}$$

and the optimal parameter vector can be found iteratively by the gradient descent method:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \Delta\boldsymbol{\theta} = \boldsymbol{\theta}_n - \alpha \mathbf{g}_\varepsilon(\boldsymbol{\theta}) = \boldsymbol{\theta}_n + \alpha E_\pi \left[(v_\pi(s) - \hat{v}(s, \boldsymbol{\theta})) \frac{d}{d\boldsymbol{\theta}} \hat{v}(s, \boldsymbol{\theta}) \right] \tag{20.74}$$

where α is the step size or learning rate, and

$$\Delta \mathbf{w} = -\alpha \mathbf{g}_\varepsilon(\mathbf{w}) = \alpha E_\pi \left[(v_\pi(s) - \hat{v}(s, \mathbf{w})) \frac{d}{d\mathbf{w}} \hat{v}(s, \mathbf{w}) \right] \tag{20.75}$$

If the method of stochastic gradient descent is used (SGD, Section6.1), i.e., the parameter $\boldsymbol{\theta}$ is updated based on only one data point at each iteration instead of the expectation of the data points, then the expectation symbol E_π can be dropped.

As the true value function $v_\pi(s)$, the expected return, is unknown, it needs to be estimated by either the actual return G_t at each state $s = s_t$ based on

the MC method, or $r_{t+1} + \gamma\hat{v}_\pi(s', \boldsymbol{\theta})$ in terms of the immediate reward r_{t+1} and the estimated value $\hat{v}_\pi(s', \boldsymbol{\theta})$ of the next state s' based on the TD method. Such an approach can be considered as a supervised learning based on labeled training data set: $\{(s_t, G_t),\ \forall t\}$ for the MC method, or $\{(s_t, r_{t+1},\ \forall t\}$ for the TD method.

Here we consider the special case where the state value $v_\pi(s^n)$ of each of the $N = |S|$ states is approximated by a linear regression function:

$$\hat{v}(s^n, \mathbf{w}) = \sum_{i=1}^{d} x_i(s^n) w_i = \mathbf{w}^T \mathbf{x}(s^n) = \mathbf{w}^T \mathbf{x}_n, \qquad n = 1, \cdots, N \tag{20.76}$$

Here the generic symbol $\boldsymbol{\theta}$ as the parameter of the regression is replaced by $\mathbf{w}$, representing specifically the weights of the linear function. This model is based on the assumption that the value function of each state s^n can be indeed approximated as a linear combination of a set of features as the components in the d-dimensional feature vector $\mathbf{x}_n = \mathbf{x}(s^n) = [x_1(s^n), \cdots, x_d(s^n)]^T$, which are hand picked or designed to properly represent the specific problem at hand.

As a special example, we can represent each of the $N = |S|$ states by a N dimensional binary feature vector $\mathbf{x}_n = \mathbf{x}(s^n) = [x_1(s^n), \cdots, x_N(s^n)]^T$, where $x_i(s^n)$ is 1 if $i = n$ but 0 otherwise. Then the weight vector becomes $\mathbf{w} = [w_1, \cdots, w_N]^T = [v_\pi(s^1), \cdots, v_\pi(s^n)]^T$, and we have $\hat{v}_\pi(s^n, \mathbf{w}) = \mathbf{w}^T \mathbf{x}_n = v_\pi(s^n)$. In this case, each state s is represented by its value function $v_\pi(s)$, the same as in all previous discussions.

The objective function for the mean squared error of such a linear approximating function is

$$\varepsilon(\mathbf{w}) = \frac{1}{2} E_\pi[(v_\pi(s) - \hat{v}(s, \mathbf{w}))^2] = \frac{1}{2} E_\pi[(v_\pi(s) - \mathbf{w}^T \mathbf{x}(s))^2] \tag{20.77}$$

If all data points in terms of state s represented by $\mathbf{x}(s)$ and the corresponding return $G(s)$ as a sample of the true value $v_\pi(s)$ have already been collected, then the value function approximation can be carried out as an off-line algorithm in a batch manner, the same as in the linear least squares regression discussed in Chapter 5, and the optimal weight vector $\mathbf{w}^*$ that minimizes the squared error

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{n=1}^{N} [G(s^n) - \mathbf{w}^T \mathbf{x}_n]^2 \tag{20.78}$$

can be found by solving a linear equation system $\mathbf{X}^T \mathbf{w} = \mathbf{g}$, where $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ is a $d \times N$ matrix containing all N states each represented by $\mathbf{x}_n$, and vector $\mathbf{g} = [G(s^1), \cdots, G(s^n)]^T$ contains the corresponding returns. As we assume the number of states is greater than the number of the model parameters, i.e., $N \geq d$, this is an over-constrained system with the least squares so1ution (Eq. (5.10)):

$$\mathbf{w}^* = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{g} = (\mathbf{X}^T)^{-}\mathbf{g} \tag{20.79}$$

However, as reinforcement learning is typically an online process, the parameter $\mathbf{w}$ for the approximator needs to be modified in real-time whenever a new piece of data becomes available while sampling the environment. Then we can use the gradient descent method, based on the gradient vector of $\varepsilon(\mathbf{w})$:

$$\begin{aligned}\mathbf{g}_\varepsilon(\mathbf{w}) &= \frac{d}{d\mathbf{w}}\varepsilon(\mathbf{w}) = -E_\pi(v_\pi(s) - \hat{v}(s,\mathbf{w}))\frac{d}{d\mathbf{w}}\hat{v}(s,\mathbf{w}) \\ &= -E_\pi(v_\pi(s) - \mathbf{w}^T\mathbf{x}(s))\mathbf{x}(s)\end{aligned} \tag{20.80}$$

and find the optimal parameter $\mathbf{w}^*$ iteratively:

$$\begin{aligned}\mathbf{w}_{n+1} &= \mathbf{w}_n + \Delta\mathbf{w} = \mathbf{w}_n + \alpha E_\pi\left[v_\pi(s) - \hat{v}(s,\mathbf{w})\right]\frac{d}{d\mathbf{w}}\hat{v}(s,\mathbf{w}) \\ &= \mathbf{w}_n + \alpha E_\pi\left[v_\pi(s) - \mathbf{w}^T\mathbf{x}(s)\right]\mathbf{x}(s)\end{aligned} \tag{20.81}$$

with increment

$$\Delta\mathbf{w} = \alpha E_\pi\left[v_\pi(s) - \hat{v}(s,\mathbf{w})\right]\frac{d}{d\mathbf{w}}\hat{v}(s,\mathbf{w}) = \alpha E_\pi\left[v_\pi(s) - \mathbf{w}_n^T\mathbf{x}(s)\right]\mathbf{x}(s) \tag{20.82}$$

If stochastic gradient descent (SGD) is used, then $\mathbf{w}$ is updated whenever a new data point $\mathbf{x}(s_t)$ is available at a new step t in the sampling process, and the expectation symbol E_π can be dropped. The expectation of the estimated $\mathbf{w}$ by this SGD method is the same as that obtained by full GD method. This iteration will convergence to the global minimum of the objective function $\varepsilon(\mathbf{w})$ in Eq. (20.77), as it is a quadratic function with only one minimum which is global.

As the state value function $v_\pi(s)$ is unknown, it needs to be estimated in one of several different ways, similar to the corresponding algorithms discussed before.

- **MC method**

  The value function $v_\pi(s)$ is estimated by the actual return G_t for each state s_t visited in an episode, obtained only at the end of each episode while sampling the environment:

$$\Delta\mathbf{w} = \alpha[G_t - \hat{v}(s_t,\mathbf{w})]\frac{d}{d\mathbf{w}}\hat{v}(s_t,\mathbf{w}) = \alpha[G_t - \mathbf{w}^T\mathbf{x}(s_t)]\mathbf{x}(s_t) \tag{20.83}$$

  Here is the pseudo code for the algorithm based on the MC method, similar to that for value evaluation algorithms discussed previously:

  Initialize $\mathbf{w} = \mathbf{0}$
  loop (for each episode)
    run current episode to the end to get $G_t,\ t = 1, \cdots, T$
    for $t = 1, \cdots, T$
      if s_t is visited the first time
        $\mathbf{w} = \mathbf{w} + \alpha[G_t - \mathbf{w}^T\mathbf{x}(s_t)]\mathbf{x}(s_t)$
      end if
    end for
  end loop

- **TD(0) method**

  The value function $v_\pi(s)$ is estimated by the TD target, the sum of the immediate reward r_{t+1} and the approximated value of the next state s_{t+1}, at each time step of each episode while sampling the environment:

$$\Delta \mathbf{w} = \alpha[r_{t+1} + \gamma \mathbf{w}^T \mathbf{x}(s_{t+1}) - \mathbf{w}^T \mathbf{x}(s_t)]\mathbf{x}(s_t) \tag{20.84}$$

  Here is the pseudo code for the algorithm based on the TD method, similar to that for value evaluation algorithms discussed previously:

  Initialize $\mathbf{w} = \mathbf{0}$
  loop (for each episode)
    for each time step in current episode
      take action $a = \pi(s)$, get reward r and next state s'
      $\mathbf{w} = \mathbf{w} + \alpha[r + \gamma \mathbf{w}^T \mathbf{x}(s') - \mathbf{w}^T \mathbf{x}(s)]\mathbf{x}(s)$
    end for
  end loop

- **TD(λ) method**

  The value function $v_\pi(s)$ is estimated based on the λ-return G_t^λ given in Eq. (20.62). As discussed before, TD(λ) has two flavors:

  – Forward view: same as MC method, except the true return G_t is replaced by λ-return G_t^λ available only at the end of each episode:

$$\Delta \mathbf{w} = \alpha[G_t^\lambda - \mathbf{w}^T \mathbf{x}(s_t)]\mathbf{x}(s_t) \tag{20.85}$$

  – Backward view: similar to TD(0), this is based on the immediate reward r_{t+1} available at every step of each episode:

$$\delta_t = \alpha[r_{t+1} + \gamma \mathbf{w}^T \mathbf{x}(s_{t+1}) - \mathbf{w}^T \mathbf{x}(s_t)]\mathbf{x}(s_t) \tag{20.86}$$

  Here the eligibility trace becomes a vector, which is initialized to zero at the beginning of the episode, and then updated by the gradient of the estimated value function $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$ (the components of $\mathbf{e}$ are updated differently instead all by 1):

$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t-1} + \frac{d}{d\mathbf{w}} \mathbf{w}^T \mathbf{x}(s_t) = \gamma \lambda \mathbf{e}_{t-1} + \mathbf{x}(s_t) \tag{20.87}$$

  while the weight vector is still updated the same as before:

$$\Delta \mathbf{w} = \alpha \delta_t \mathbf{e}_t \tag{20.88}$$

Given a policy π for an MDP, we also define a probability distribution $d(s)$ over all states visited according to π, satisfying

$$\sum_s d(s) = 1 \tag{20.89}$$

and the following balance equation:

$$d(s') = \sum_s \sum_a \pi(a|s) p(s'|s, a) d(s) \tag{20.90}$$

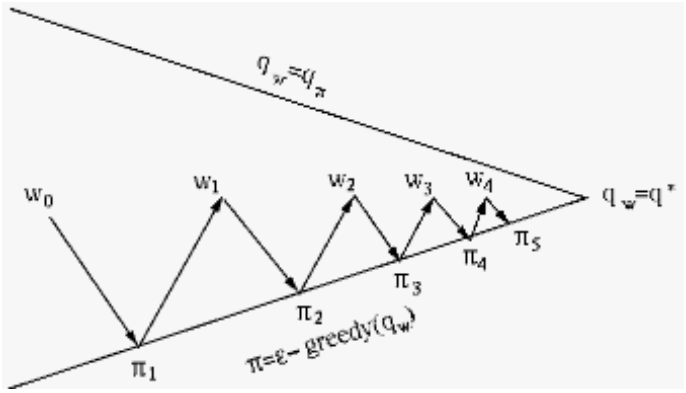


Figure 20.11 Policy Iteration Based on Approximated Value Functions

As both $d(s')$ and $d(s)$ represent the same distribution, they must be identical.

Given the probability distribution $d(s)$, the mean squared error of the value function approximation for a policy π can be expressed as

$$MSE(\mathbf{w}) = \sum_s d(s)[v_\pi(s) - \hat{v}_\pi(s, \mathbf{w})]^2 \tag{20.91}$$

It can be shown that the MC method for the linear value function approximation will converge to the optimal weight vector $\mathbf{w}_{MC}$ that minimizes the mean squared error above:

$$MSE(\mathbf{w}_{MC}) = \min_{\mathbf{w}} \sum_s d(s)[v_\pi(s) - \hat{v}_\pi(s, \mathbf{w})]^2 \tag{20.92}$$

20.5 Control Based on Function Approximation

Model-free control can be carried out based on the state action value function $\hat{q}(s, a, \mathbf{w})$ approximated as discussed in the previous section, by the approach of general policy iteration, as illustrated in Fig. 20.11. This is similar to the algorithms for model-free control illustrated in Fig. 20.5, with the only difference that the action-value function $q_\pi(s, a)$ is now replaced by the parameter $\mathbf{w}$ of the linear model of the action value function:

$$\hat{q}(s, a, \mathbf{w}) = \sum_n w_n x(s, a) = \mathbf{w}^T \mathbf{x}(s, a) \tag{20.93}$$

where $\mathbf{x}(s, a)$ is the feature vector for the state-action pair (s, a). We need to find the optimal parameter $\mathbf{w}$ that minimizes the objective function, here specifically the mean square error of the approximation:

$$\varepsilon(\mathbf{w}) = \frac{1}{2} E_\pi[(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w}))^2] = \frac{1}{2} E_\pi[(q_\pi(s, a) - \mathbf{w}^T \mathbf{x}(s, a))^2] \tag{20.94}$$

with gradient vector:

$$\begin{aligned}\mathbf{g}_\varepsilon(\mathbf{w}) &= \frac{d}{d\mathbf{w}}\varepsilon(\mathbf{w}) = -E_\pi\left[q_\pi(s,a) - \hat{q}(s,a,\mathbf{w})\right]\frac{d}{d\mathbf{w}}\hat{q}(s,a,\mathbf{w}) \\ &= -E_\pi\left[(q_\pi(s,a) - \mathbf{w}^T\mathbf{x}(s,a))\mathbf{x}(s,a)\right] \end{aligned} \tag{20.95}$$

If the stochastic gradient descent method is used based on a single sample of the action value $q_\pi(s,a)$, instead of its expectation, then the expectation symbol E_π can be dropped. The optimal weight vector $\mathbf{w}^*$ that minimizes $\varepsilon(\mathbf{w})$ in Eq. (20.94) can be learned iteratively:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \Delta\mathbf{w} = \mathbf{w}_t - \alpha\mathbf{g}_\varepsilon(\mathbf{w}) \\ &= \mathbf{w}_t + \alpha\left[(q_\pi(s_t,a_t) - \hat{q}_\pi(s_t,a_t,\mathbf{w}))\frac{d}{d\mathbf{w}}\hat{q}_\pi(s_t,a_t,\mathbf{w})\right] \\ &= \mathbf{w}_t + \alpha\left[(q_\pi(s_t,a_t) - \mathbf{w}_t^T\mathbf{x}(s_t,a_t))\mathbf{x}(s_t,a_t)\right] \end{aligned} \tag{20.96}$$

where $\Delta\mathbf{w}$ is the increment of the update:

$$\begin{aligned}\Delta\mathbf{w} = -\alpha\mathbf{g}_\varepsilon(\mathbf{w}) &= \alpha[q_\pi(s_t,a_t) - \hat{q}_\pi(s_t,a_t,\mathbf{w})]\frac{d}{d\mathbf{w}}\hat{q}_\pi(s_t,a_t,\mathbf{w}) \\ &= \alpha[q_\pi(s_t,a_t) - \mathbf{w}_t^T\mathbf{x}(s_t,a_t)]\mathbf{x}(s_t,a_t) \end{aligned} \tag{20.97}$$

As the true Q-value $q_\pi(s,a)$ in the expression is unknown, it needs to be estimated by some target depending on the specific methods used:

- **MC method:**

  The true $q_\pi(s,a)$ is replaced by the sample return G_t as the target, obtained at the end of each episode:

$$\Delta\mathbf{w} = \alpha[G_t - \hat{q}(s_t,a_t,\mathbf{w})]\frac{d}{d\mathbf{w}}\hat{q}(s_t,a_t,\mathbf{w}) = \alpha[G_t - \mathbf{w}_t^T\mathbf{x}(s_t,a_t)]\mathbf{x}(s_t,a_t) \tag{20.98}$$

- **TD(0) method:**

  The action value function $q_\pi(s,a)$ is replaced by the TD target, the sum of the immediate reward, available at each step of each episode, and the approximated action value of the next state s_{t+1}:

  – SARSA (on-policy):

$$\begin{aligned}\Delta\mathbf{w} &= \alpha[r_{t+1} + \gamma\hat{q}(s_{t+1},a_{t+1},\mathbf{w}) - \hat{q}(s_t,a_t,\mathbf{w})]\frac{d}{d\mathbf{w}}\hat{q}(s_t,a_t,\mathbf{w}) \\ &= \alpha[r_{t+1} + \gamma\mathbf{w}^T\mathbf{x}(s_{t+1},a_{t+1}) - \mathbf{w}_t^T\mathbf{x}(s_t,a_t)]\mathbf{x}(s_t,a_t) \end{aligned} \tag{20.99}$$

  Following Eq. (20.71), the TD error is defined as:

$$\delta_t = r_{t+1} + \gamma\hat{q}(s_{t+1},a_{t+1},\mathbf{w}) - \hat{q}(s_t,a_t,\mathbf{w}) \tag{20.100}$$

  then we get

$$\Delta\mathbf{w} = \alpha\delta_t\frac{d}{d\mathbf{w}}\hat{q}(s_t,a_t,\mathbf{w}) \tag{20.101}$$

– Q-learning (off-policy):

$$\begin{aligned}\Delta\mathbf{w} &= \alpha[r_{t+1} + \gamma \max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}) - \hat{q}(s_t, a_t, \mathbf{w})]\frac{d}{d\mathbf{w}}\hat{q}(s_t, a_t, \mathbf{w}) \\ &= \alpha[r_{t+1} + \gamma \max_{a'} \mathbf{w}^T\mathbf{x}(s_{t+1}, a') - \mathbf{w}_t^T\mathbf{x}(s_t, a_t)]\mathbf{x}(s, a) \quad (20.102)\end{aligned}$$

- **TD(λ) method:**

  In the forward-view version of the TD(λ) method, the action function $q_\pi(s, a)$ is approximated by λ-return G_t^λ as the target, available only at the end of each episode:

$$\Delta\mathbf{w} = \alpha[G_t^\lambda - \mathbf{w}_t^T\mathbf{x}(s_t, a_t)]\mathbf{x}(s_t, a_t) \quad (20.103)$$

  In the backward-view version of the TD(λ) method, the eligibility trace in Eq. (20.87) becomes:

$$\mathbf{e}_t = \gamma\lambda\mathbf{e}_{t-1} + \frac{d}{d\mathbf{w}}\hat{q}(s_t, a_t, \mathbf{w}) \quad (20.104)$$

  and the weight vector is updated as in Eq. (20.88):

$$\Delta\mathbf{w} = \alpha\delta_t\mathbf{e}_t \quad (20.105)$$

  where δ_t is the TD error for the backward view of TD(λ) first given in Eq. (20.71), but now with the state value $v_\pi(s)$) replaced by the estimated action value $\hat{q}(s, a, \mathbf{w})$:

$$\delta_t = (r_{t+1} + \gamma\hat{q}_\pi(s_{t+1}, a_{t+1}) - \hat{q}_\pi(s_t, a_t)) \quad (20.106)$$

In summary, here are the conceptual (not necessarily algorithmic) steps for the general model-free control based on approximated action-value function:

- Learn parameter $\mathbf{w}$ as in Eq. (20.96),
- Get the Q-values as in Eq. (20.93)
- Obtain the policy by ϵ-greedy approach as in Eq. (20.42)

These steps are also illustrated below:

$$\text{Training of } \mathbf{w} \Longrightarrow \text{Q-value} \Longrightarrow \text{Policy } \pi \quad (20.107)$$

20.6 Deep Q-learning

In the previous section, the Q-values at a given state are approximated by a linear regression function $\hat{q}(s, a, \mathbf{w})$ (Eq. (20.93).) parameterized by the weight vector $\mathbf{w}$. However, when applied to a real world problem with large numbers of states and actions, this simple linear function may not be able to model the potentially highly complex nonlinear relationship between the state-action pairs and the value functions. In this case, a more sophisticated nonlinear model may be needed, such as a deep learning networks considered in Section 18.4 used for modeling complex nonlinear relationship $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{W})$ between its input $\mathbf{x}$ and

output $\mathbf{y}$. When applied to reinforcement learning, such a deep learning network takes a given state s as input, and generates the approximated Q-values $\hat{q}(s, a, \mathbf{W})$ for all $|A|$ actions available at the state as output. Here $\mathbf{W}$ represents all weights in the deep network, while vector $\mathbf{w}$ contains all weights of a linear regression function $\hat{q}(s, a, \mathbf{w})$. This deep learning network in reinforcement learning is called the *deep Q-learning network (DQN)*, while all previously considered methods such as the methods of tabular Q-learning and linear function approximation are considered shallow. The three methods are illustrated in Fig. 20.12.

As we know, the method of tabular Q-learning models the environment by the Q-table of size $|A||S|$ for all actions at all states, which may become impractically large when both $|S|$ and $|A|$ are large. This difficulty is avoided in DQN method, as well as in regression function approximation, which estimates the action value functions $\hat{q}(s, a)$ without exhausting all $|A||S|$ cases. The cases not ever seen during training are approximated by interpolation based on those that have been seen. Also, the DQN method is superior to function approximation due to its multi-layer nonlinear learning capability. Moreover, while both methods take the current state s as input, function approximation is based on a small set of carefully handcrafted states as input, but the input to the DQN can be in some raw data with potentially high dimensionality extracted directly from the environment. For example, in a board game problem, it is very difficult to come up with a set of well defined states that characterizes all possible situations in the game. But for the DQN, the current state can be represented simply by the image of the board, of which all pixel values are directly fed into the input layer of the DQN (similar to the image input to an CNN considered in Section 18.4).

As a supervised learning method, the DQN is trained in the same way as a back-propagation network. The weights in $\mathbf{W}$ as the model parameter are iteratively updated by gradient descent, to gradually reduce the error, the difference between the actual output $\hat{q}(s, a, \mathbf{W})$ of the DQN and the desired target output, obtained by the Bellman equation (Eq. (20.33)) based on data collected while sampling the environment, including the current state s, the action a, the next state s', and the reward r, collectively treated as an *experience tuple* $e = (s, a, s', r)$:

$$q(s, \mathbf{W}') = r + \gamma \max_{a'} q(s', a', \mathbf{W}') \tag{20.108}$$

Here the second term is the highest output value from the DQN based on the previous estimated weights, denoted by $\mathbf{W}'$. We denote the target $q(s, \mathbf{W}')$ simply as q as it is treated as a constant while updating $\mathbf{W}$. Now the squared error between the DQN output and the target can be found is:

$$\varepsilon(\mathbf{W}) = \frac{1}{2}\left[q - \hat{q}(s, a, \mathbf{W})\right]^2 = \frac{1}{2}\left[r + \gamma \max_{a'} q(s', a', \mathbf{W}') - \hat{q}(s, a, \mathbf{W})\right]^2 \tag{20.109}$$

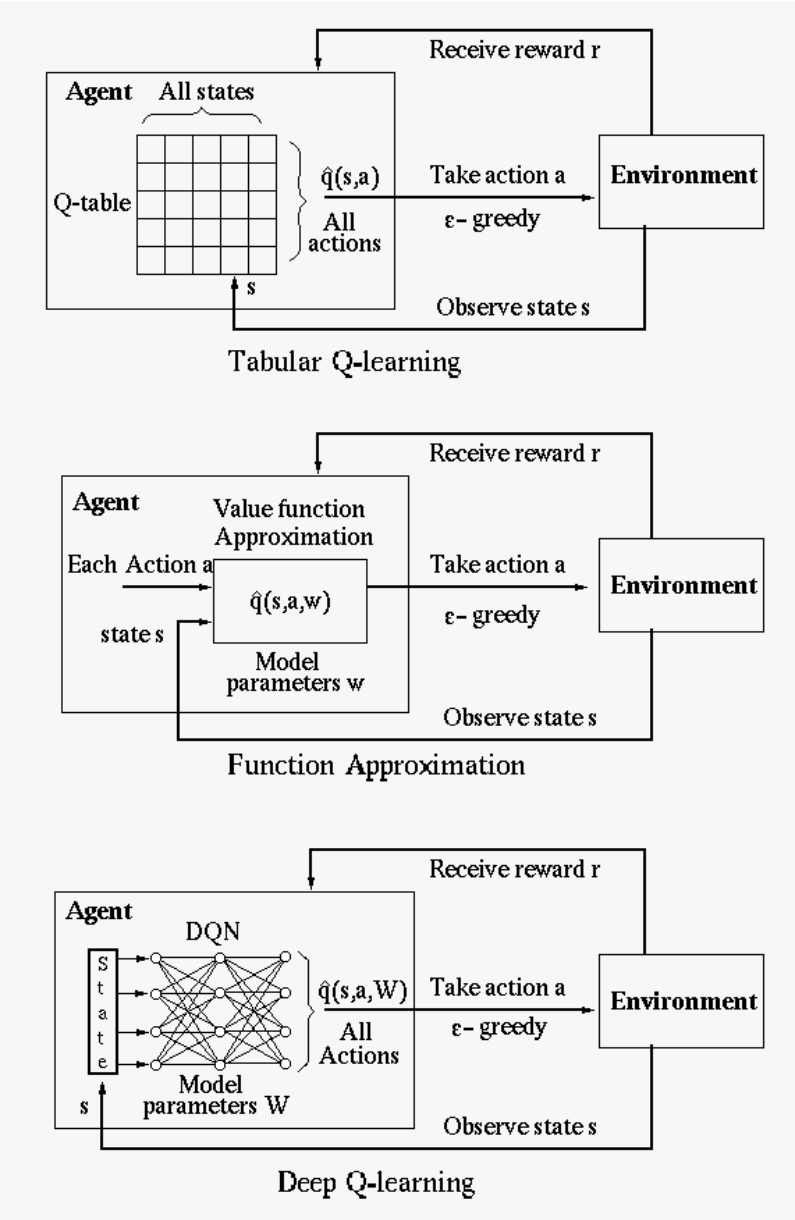


Figure 20.12 Three Q-Learning Methods: Tabular, Function Approximation, and Deep Q-Learning

and its gradient

$$\begin{aligned}\mathbf{g}_{\varepsilon}(\mathbf{W}) = \frac{d}{d\mathbf{W}}\varepsilon(\mathbf{W}) &= -\left[q(s,\mathbf{W}') - \hat{q}(s,a,\mathbf{W})\right]\frac{d}{d\mathbf{W}}\hat{q}(s,a,\mathbf{W}) \\ &= -\left[r + \gamma \max_{a'} q(s',a',\mathbf{W}') - \hat{q}(s,a,\mathbf{W})\right]\frac{d}{d\mathbf{W}}\hat{q}(s,a,\mathbf{W}) \quad (20.110)\end{aligned}$$

The weights in $\mathbf{W}$ are updated by gradient descent:

$$\begin{aligned}\mathbf{W}_{n+1} &= \mathbf{W}_n - \alpha \mathbf{g}_\varepsilon(\mathbf{W}) \\ &= \mathbf{W}_n + \alpha \left[r + \gamma \max_{a'} q(s', a', \mathbf{W}') - \hat{q}(s, a, \mathbf{W}) \right] \frac{d}{d\mathbf{W}} \hat{q}(s, a, \mathbf{W}) \end{aligned} \quad (20.111)$$

For this deep Q-learning method there are two issues that need to be further addressed.

- First, to update $\mathbf{W}$ by Eq. (20.111) we need to estimate the target by the Bellman equation in Eq. (20.108). In this bootstrapping process, the resulting target $q(s, \mathbf{W}')$ depends on the previous estimate $q(s', a', \mathbf{W}')$, which in turn depends on the previous $\mathbf{W}'$, i.e., the iterative training process is actually chasing a moving target. This is different from the training of a conventional back-propagation network, where the desired output, the target, is fixed. As the consequence, the training iteration of the DQN may be unstable, even diverging.

  This problem is addressed by including a second network, called the target network and parameterized by $\mathbf{W}'$. The output $q'(s', a', \mathbf{W}')$ of this target network is used as the targets for training the first network, of which the output $q(s, a, \mathbf{W})$ is used for the state transition while sampling the environment. To avoid chasing a moving target, here $\mathbf{W}'$ of the target network remain unchanged for some period of C steps, while $\mathbf{W}$ of the first network is updated. At the end of the C-step period, the updated $\mathbf{W}$ is copied over to the target network to replace its $\mathbf{W}'$.
- Second, during the sequential sampling of the environment step by step, the Q values obtained for the current state depends on those collected in the previous state, i.e., they are not i.i.d. samples as they are correlated. Consequently the training process may be biased. This issue is similar to that considered in Subsection 20.3.1, where we only use the data collected during the first visit, instead of every visit, to a state while sampling the environment.

  This problem is addressed by a method called *experience replay* based on a buffer $D = \{e_1, \cdots, e_N\}$ containing the agent's previous experience tuples collected in the latest N steps while sampling the environment. The DQN is then trained based on a minibatch of such experiences randomly selected from the buffer D. As these experiences are in random order and no longer correlated, the training is unbiased.

Once the DQN has been trained, the agent can form the optimal policy by taking the action a corresponding to the maximum $q(s, a)$ out of all outputs of the DQN for each state s as the input.

Here is the pseudo code for the deep Q-learning method:

Input: the states and their corresponding rewards in the environment (e.g., images of game board and the scores)

Output: Q-values for all actions in states
Initialize replay buffer D for N experiences
Initialize $\mathbf{W} = \mathbf{W}'$ for both the target network and DQN
loop (for each episode)
    for each time step in current episode
        Take action a at current state s based on ϵ-greedy policy:

$$a = \begin{cases} \text{random action} & Prob = \epsilon \\ \arg\max_a q(s, a, \mathbf{W}) & Prob = 1 - \epsilon \end{cases}$$

        Get reward r and next state s'
        Store a (s, a, s', r) in D
        Move to next state:$s = s'$
        For each experience $(s, a, s', , r)$ in a random minibatch from D
        Find target: :

$$q = \begin{cases} r & \text{if } s' \text{ is a terminal state} \\ r + \gamma \max_{a'} q'(s', a', \mathbf{W}') \end{cases}$$

        Update $\mathbf{W}$ by gradient descent based on gradient $\mathbf{g}_\epsilon(\mathbf{W})$ of squared error $\varepsilon(\mathbf{W}) = (q - q(, s, a, \mathbf{W}))^2$
        Update weights: $\mathbf{W}' = \mathbf{W}$ (i.e., $q' = q$) every C steps
    end for
end loop

20.7 Policy Gradient Methods

All RL algorithms previously considered are based on either state or action value function and the policy is indirectly derived from them by greedy or ϵ-greedy method. However, as the ultimate goal of an RL problem is to find the optimal policy that maximizes the return, it can also be considered as an optimization problem to directly find the optimal policy that maximizes an objective function representing the total return, which can be addressed by the gradient ascent method, as discussed below.

In previous methods we approximated the state value function $v_\pi(s)$ or action value function $q_\pi(s, a)$ by a function $\hat{v}_\pi(s, \mathbf{w})$ or $\hat{q}_\pi(s, a, \mathbf{w})$, of which the parameter $\mathbf{w}$ is obtained by sampling the environment, as a supervised learning process. Now we will directly construct a model of the policy as a function parameterized by $\boldsymbol{\theta}$:

$$\pi_\theta(a|s) = P(a|s, \boldsymbol{\theta}) \tag{20.112}$$

where $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_d]^T$ is a vector composed of d parameters of the policy model. Once $\boldsymbol{\theta}$ is available, the probability for any action at any state is determined. This parameterized policy model is suitable in cases where the numbers of states and actions are large or even continuous.

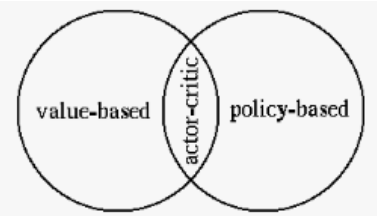


Figure 20.13 Value-Based, Policy-Based, and Actor-Critic

As a specific example, the policy model can be based on the soft-max function:

$$\pi_\theta(a|s) = P(a|s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{a'} e^{h(s,a',\boldsymbol{\theta})}} \tag{20.113}$$

satisfying $\sum_a \pi_\theta(a|s) = 1$. Here the summation is over all possible actions in state s, and $h(s, a, \boldsymbol{\theta})$ is the *preference* of action a in state s, which can be a parameterized function such as a simple linear function $h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathbf{x}(s, a)$, or a neural network with weights as the parameters in $\boldsymbol{\theta}$, the same as how the value functions are approximated in Section 20.4. According to this policy, an action with higher preference $h(s, a, \boldsymbol{\theta})$ will have a higher probability to be chosen at state s.

Different from how we find the parameter $\mathbf{w}$ of $\hat{q}(s, a, \mathbf{w})$ by minimizing the objective function $\varepsilon(\mathbf{w})$ in Eq. (20.94) for the mean squared error between the value function $q_\pi(s, a)$ and its model $\hat{q}_\pi(s, a, \mathbf{w})$, here we find the parameter $\boldsymbol{\theta}$ of $\pi_\theta(a|s)$ by maximizing an objective function $J(\boldsymbol{\theta})$ representing the value function, the expected return, under the policy, which is to be minimized by gradient ascent. Such a method is therefore called a *policy gradient method*.

The value-function based methods considered previously and the policy-based methods considered here are summarized below, together with the*actor-critic method*, as the combination of the two. These methods are also illustrated in Fig. 20.13

- Value-based: the policy is derived by the greedy or ϵ-greedy method based on the value function learned by sampling the environment as a supervised training process.
- Policy-based: the policy is directly learned without explicitly value function estimation.
- Actor-Critic: parameters for both value function model and policy model are learned simultaneously.

A policy-based method can learn stochastic policies effectively in high-dimensional or continuous action space with better convergence properties, but it may get stuck at a local optimum, and has high variance.

How good a policy is may be measured by different objective functions related

to the values or rewards associated with the policy being evaluated, depending on the environment of the specific problem:

- In episodic environments with finite horizon, we can use the value, the expected return of the start state s_0:

$$J_1(\boldsymbol{\theta}) = v_{\pi_\theta}(s_0) \tag{20.114}$$

- In continuing (online) environments with infinite horizon, we can use the average value

$$J_{avv}(\boldsymbol{\theta}) = \sum_s d_{\pi_\theta}(s) v_{\pi_\theta}(s) \tag{20.115}$$

  where $d_{\pi_\theta}(s)$ is the stationary distribution of all states under policy $\pi(a|s,\theta)$.

We can find the optimal parameter $\boldsymbol{\theta}^*$ for the policy model $\pi_\theta(a|s)$ by solving the maximization problem:

$$\theta^* = \arg\max_\theta J(\boldsymbol{\theta}) \tag{20.116}$$

based on the gradient of $J(\boldsymbol{\theta})$:

$$\bigtriangledown_\theta J(\theta) = \frac{d}{d\boldsymbol{\theta}} J(\theta) = \left[\frac{\partial J}{\partial \theta_1}, \cdots, \frac{\partial J}{\partial \theta_n}\right]^T \tag{20.117}$$

by the iterative gradient ascent method:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta} = \boldsymbol{\theta}_t + \alpha \bigtriangledown_\theta J(\boldsymbol{\theta}) \tag{20.118}$$

Here we use t as the index of the iteration based on the assumption that a new sample point is available at every time step of an episode while sampling the environment following policy $\pi_\theta(a|s)$.

While it is conceptually straight forward to see how the optimal parameter $\boldsymbol{\theta}^*$ that maximizes $J(\boldsymbol{\theta})$ can be found by the gradient ascent method, it is not easy to actually obtain the gradient $\bigtriangledown_\theta J(\boldsymbol{\theta})$, which depends on not only what actions to take at the states directly determined by the policy $\pi_\theta(a|s)$, but also how the states are distributed under the policy in an unknown environment. This challenge is addressed by the following *policy gradient theorem*, stating that the gradient of $J(\boldsymbol{\theta})$ can be calculated as shown below without explicit derivatives of the state distribution:

$$\bigtriangledown_\theta J(\boldsymbol{\theta}) = E_{\pi_\theta}\left[\bigtriangledown_\theta \ln \pi_\theta(a|s)\, q_{\pi_\theta}(s,a)\right] \tag{20.119}$$

Proof of policy gradient theorem:

Consider the derivative of the value function $v_{\pi_\theta} = \sum_a \pi_\theta(a|s)\ q_{\pi_\theta}(s,a)$ (in Eq. (20.21)):

$$
\begin{aligned}
\bigtriangledown_\theta v_{\pi_\theta}(s) &= \bigtriangledown_\theta \left(\sum_a \pi_\theta(a|s) q_{\pi_\theta}(s,a) \right) \\
&= \sum_a \left[\bigtriangledown_\theta \pi_\theta(a|s)\ q_{\pi_\theta}(s,a) + \pi_\theta(a|s)\ \bigtriangledown_\theta\ q_{\pi_\theta}(s,a) \right] \\
&\overset{1}{=} \sum_a \left[\bigtriangledown_\theta \pi_\theta(a|s)\ q_{\pi_\theta}(s,a) + \pi_\theta(a|s) \bigtriangledown_\theta \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) v_{\pi_\theta}(s') \right) \right] \\
&\overset{2}{=} \sum_a \left[\bigtriangledown_\theta \pi_\theta(a|s)\ q_{\pi_\theta}(s,a) + \pi_\theta(a|s) \gamma \sum_{s'} P(s'|s,a) \bigtriangledown_\theta v_{\pi_\theta}(s')) \right] \\
&= \sum_a \bigtriangledown_\theta \pi_\theta(a|s)\ q_{\pi_\theta}(s,a) + \sum_a \pi_\theta(a|s) \gamma \sum_{s'} P(s'|s,a) \bigtriangledown_\theta v_{\pi_\theta}(s') \\
&\overset{3}{=} \phi(s) + \sum_a \pi_\theta(a|s) \gamma \sum_{s'} P(s'|s,a) \bigtriangledown_\theta v_{\pi_\theta}(s')
\end{aligned} \tag{20.120}
$$

where the labeled equal signs are due to the following:

1. Eq. (20.22)
2. $r(s,a)$ is not a function of θ
3. We have defined

$$
\phi(s) = \sum_a \bigtriangledown_\theta \pi_\theta(a|s)\ q_{\pi_\theta}(s,a) \tag{20.121}
$$

Eq. (20.120) is a recursion by which $\bigtriangledown_\theta v_{\pi_\theta}(s)$ is expressed as a function in terms $\bigtriangledown_\theta v_{\pi_\theta}(s')$.

We further consider the probability of transitioning from state s to a state x after k steps following $\pi(a|s)$:

$$
s \overset{\pi(a|s)}{\longrightarrow} s' \overset{\pi(a|s')}{\longrightarrow} s'' \overset{\pi(a|s'')}{\longrightarrow} \cdots \overset{\pi(a|s^{k-1})}{\longrightarrow} s^k = x \tag{20.122}
$$

denoted by $P_\pi(s \to x, k)$ with the following properties:

- $$
P_\pi(s \to s, 0) = 1 \tag{20.123}
$$

- $$
P_\pi(s \to s', 1) = \sum_a \pi(a|s) P(s'|s,a) \tag{20.124}
$$

- $$
\sum_{s'} P_\pi(s \to s', 1) = 1 \tag{20.125}
$$

- $$
P_\pi(s \to s'', 2) = P_\pi(s \to s', 1) P_\pi(s' \to s'', 1) \tag{20.126}
$$

•

$$P_\pi(s \to x, k+1) = \sum_{s'} P_\pi(s \to s', k) P_\pi(s' \to x, 1) \tag{20.127}$$

Continuing Eq. (20.120) we keep rolling out the recursion of $\bigtriangledown_\theta v_{\pi_\theta}(s')$ and get:

$$\begin{aligned}
\bigtriangledown_\theta v_{\pi_\theta}(s) &= \phi(s) + \gamma \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \bigtriangledown_\theta v_{\pi_\theta}(s') \\
&\overset{1}{=} \phi(s) + \gamma \sum_{s'} P_{\pi_\theta}(s \to s', 1) \bigtriangledown_\theta v_{\pi_\theta}(s') \\
&\overset{2}{=} \phi(s) + \gamma \sum_{s'} P_{\pi_\theta}(s \to s', 1) \left[\phi(s') + \gamma \sum_{s''} P_{\pi_\theta}(s' \to s'', 1) \bigtriangledown_\theta v_{\pi_\theta}(s'') \right] \\
&\overset{3}{=} \phi(s) + \gamma \sum_{s'} P_{\pi_\theta}(s \to s', 1) \phi(s') + \gamma^2 \sum_{s''} \sum_{s'} P_{\pi_\theta}(s \to s', 1) P_{\pi_\theta}(s' \to s'', 1) \bigtriangledown_\theta v_{\pi_\theta}(s'') \\
&\overset{4}{=} \phi(s) + \gamma \sum_{s'} P_{\pi_\theta}(s \to s', 1) \phi(s') + \gamma^2 \sum_{s''} P_{\pi_\theta}(s \to s'', 2) \bigtriangledown_\theta v_{\pi_\theta}(s'') \\
&\overset{5}{=} \phi(s) + \gamma \sum_{s'} P_{\pi_\theta}(s \to s', 1) \phi(s') + \gamma^2 \sum_{s''} P_{\pi_\theta}(s \to s'', 2) \phi(s'') + \gamma^3 \sum_{s'''} P_{\pi_\theta}(s \to s''', 2) \bigtriangledown \\
&= \cdots\cdots \overset{6}{=} \sum_{x \in S} \sum_{k=0}^{\infty} \gamma^k P_{\pi_\theta}(s \to x, k) \phi(x)
\end{aligned}$$

where the labeled equal signs are due to the following:

1. Eq. (20.124)
2. unroll $\bigtriangledown_\theta v_{\pi_\theta}(s')$
3. Eq. (20.125)
4. Eq. (20.127)
5. unroll $\bigtriangledown_\theta v_{\pi_\theta}(s'')$
6. unrolling recursively to infinity; Eq. (20.123)

Now the gradient of the objective $J(\theta) = v_{\pi_\theta}(s_0)$ can be written as

$$\begin{aligned}
\bigtriangledown_\theta J(\theta) = \bigtriangledown_\theta v_\pi(s_0) &= \sum_{s \in S} \sum_{k=0}^{\infty} \gamma^k P_{\pi_\theta}(s_0 \to s, k)\ \phi(s) \\
&= \sum_{s \in S} \eta(s)\ \phi(s) = \left(\sum_{s'} \eta(s') \right) \sum_{s \in S} \frac{\eta(s)}{\sum_{s'} \eta(s')}\ \phi(s) \\
&\propto \sum_{s \in S} \frac{\eta(s)}{\sum_{s'} \eta(s')}\ \phi(s) = \sum_{s \in S} \mu_\pi(s)\ \sum_a \bigtriangledown_\theta \pi_\theta(a|s)\ q_{\pi_\theta}(s,a)
\end{aligned} \tag{20.129}$$

where we also defined

$$\eta(s) = \sum_{k=0}^{\infty} \gamma^k P_{\pi_\theta}(s_0 \to s, k), \tag{20.130}$$

as the sum of all probabilities for visiting state s from the start state s_0, and

$$\mu_\pi(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} \tag{20.131}$$

as a normalized version of $\eta(s)$ representing the probability distribution of visiting state s while following policy π. Here the proportionality is due to the dropping of the constant $\sum_{s'} \eta(s')$ independent of state s.

Following the similar steps, the policy gradient theorem for environment with continuous state and action spaces can be also proven:

$$\bigtriangledown_\theta J(\theta) = \int_s \mu_\pi(s) \int_a \bigtriangledown_\theta \pi_\theta(a|s)\, q_{\pi_\theta}(s,a) da\, ds \tag{20.132}$$

Now Eq. (20.129) can be further written as

$$\begin{aligned} \bigtriangledown_\theta J(\theta) = \bigtriangledown_\theta v_\pi(s_0) &\propto \sum_{s\in S} \mu_\pi(s) \sum_a \pi_\theta(a|s) \frac{\bigtriangledown_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}\, q_{\pi_\theta}(s,a) \\ &= \sum_{s\in S} \mu_\pi(s) \sum_a \pi_\theta(a|s)\ \bigtriangledown_\theta \ln \pi_\theta(a|s)\, q_{\pi_\theta}(s,a) \\ &= E_{\pi_\theta}\left[\bigtriangledown_\theta \ln \pi_\theta(a|s)\, q_{\pi_\theta}(s,a)\right] \end{aligned} \tag{20.133}$$

where E_{π_θ} denotes the expectation over all actions in each state s weighted by $\pi_\theta(a|s)$ and all states $s \in S$ weighted by $\mu_\pi(s)$.

Q.E.D.

The gradient $\bigtriangledown_\theta J(\theta)$ is now expressed as the expectation of the action value function $q_\pi(s,a)$, weighted by the gradient of the logarithm of the corresponding policy $\pi_\theta(a|s)$. Now the gradient ascent iteration in Eq. (20.118) can be further written as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \bigtriangledown_\theta J(\boldsymbol{\theta}) = \theta_t + \alpha E_{\pi_\theta}\left[q_{\pi_\theta}(s_t,a_t)\ \bigtriangledown_\theta \ln \pi_\theta(a_t|s_t)\right] \tag{20.134}$$

Now Eq. (20.118) can also be written as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \bigtriangledown_\theta J(\boldsymbol{\theta}) = \theta_t + \alpha E_{\pi_\theta}\left[(q_{\pi_\theta}(s_t,a_t) + b(s_t))\ \bigtriangledown_\theta \ln \pi_\theta(a_t|s_t)\right] \tag{20.135}$$

The expectation E_{π_θ} in Eqs. (20.134) and (20.135) can be dropped if the method of stochastic gradient ascent is used, as in all algorithms below, where each iterative step is based on only one sample data point instead of its expectation.

In particular, for a soft-max policy model as given in Eq. (20.113), we have

$$\begin{aligned}\bigtriangledown_\theta \ln \pi_\theta(a|s) &= \bigtriangledown_\theta \ln \left(\frac{e^{\boldsymbol{\theta}^T \mathbf{x}(s,a)}}{\sum_b e^{\boldsymbol{\theta}^T \mathbf{x}(s,b)}} \right) = \bigtriangledown_\theta \left(\theta^T \mathbf{x}(s,a) - \ln \sum_b e^{\theta^T \mathbf{x}(s,b)} \right) \\ &= \mathbf{x}(s,a) - \frac{\bigtriangledown_\theta \sum_b e^{\theta^T \mathbf{x}(s,b)}}{\sum_b e^{\theta^T \mathbf{x}(s,b)}} = \mathbf{x}(s,a) - \frac{\sum_b e^{\theta^T \mathbf{x}(s,b)} \mathbf{x}(s,b)}{\sum_b e^{\theta^T \mathbf{x}(s,b)}} \\ &= \mathbf{x}(s,a) - \sum_b \frac{e^{\theta^T \mathbf{x}(s,b)}}{\sum_b e^{\theta^T \mathbf{x}(s,b)}} \mathbf{x}(s,b) \\ &= \mathbf{x}(s,a) - \sum_b \pi_\theta(b|s) \mathbf{x}(s,b) \end{aligned} \tag{20.136}$$

A set of popular policy-based algorithms is listed below. They are based on either the MC or TD methods discussed before.

- **REINFORCE (MC) policy gradient**

  In this algorithm, the action value function $q_{\pi_\theta}(s_t, a_t)$ in Eq. (20.134) is replaced by the return G_t obtained at the end of each episode while sampling the environment. Now the equation becomes:

$$\theta_{t+1} = \theta_t + \alpha \gamma^t G_t \ \bigtriangledown_\theta \ln \pi_\theta(a|s) \tag{20.137}$$

  Note that the discount factor γ^t is included as the expression for $\bigtriangledown_\theta J(\theta)$ in Eq. (20.133) assumed $\gamma = 1$ for simplicity.

  Here is the pseudo code for the algorithm:

  Initialize $\pi(a|s, \theta$

  loop (for each episode)

    At the end of episode, get G_t for each state visited

    for $t = 1, \cdots, T$

      $\theta = \theta + \alpha \gamma^t G_t \bigtriangledown_\theta \ln \pi_\theta(a_t|s_t)$

  end loop

- **Actor-Critic (TD) policy gradient**

  As its name suggests, this algorithm is based on two approximation function models, the first for the policy $\pi_\theta(a|s)$ parameterized by $\boldsymbol{\theta}$, the *actor*, the same as in the REINFORCE algorithm above, and the second for the value function $\hat{v}_\pi(s, \mathbf{w})$ parameterized by $\mathbf{w}$, the *critic*, the same as in Eq. (20.96).

  In Eq. (20.135), the action value function $q_{\pi_\theta}(s_t, a_t)$ is replaced by its bootstrapping expression $r_{t+1} + \gamma \hat{v}_\pi(s', \mathbf{w})$, and the bias term $b(s)$ is replaced by the approximated value function $\hat{v}(s_t, \mathbf{w})$. Now both parameters $\mathbf{w}$ and $\boldsymbol{\theta}$ can be found iteratively at every step of an episode while sampling the environment (the TD method) by stochastic gradient method with the expectation symbol E_π dropped:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha_w \left[(r_{t+1} + \gamma \hat{v}_\pi(s', \mathbf{w}) - \hat{v}_\pi(s, \mathbf{w})) \ \bigtriangledown_w \hat{v}_\pi(s, \mathbf{w}) \right] \\ \theta_{t+1} &= \theta_t + \alpha_\theta \left[(r_{t+1} + \gamma \hat{v}_\pi(s', \mathbf{w}) - \hat{v}_\pi(s, \mathbf{w})) \ \bigtriangledown_\theta \ln \pi_\theta(a|s) \right] \end{aligned} \tag{20.138}$$

Here is the pseudo code for the algorithm:

initialize model parameters $\boldsymbol{\theta}$ and $\mathbf{w}$
initialize step sizes α_θ and α_w
loop (for each episode)
    initialize $s = s_0$, $t = 0$
    while s is not terminal (for each step)
        take action a following $\pi_\theta(a|s)$, find reward r and next state s'
        find TD error: $\delta = r + \gamma \hat{v}(s', \mathbf{w}) - \hat{v}(s, \mathbf{w})$
        $\mathbf{w} = \mathbf{w} + \alpha_w \, \delta \, \bigtriangledown \hat{v}(s, \mathbf{w})$
        $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta \gamma^t \delta \, \bigtriangledown \ln \pi_\theta(a|s)$
        $s = s'$
        $t = t + 1$
    end while
end loop

- **Backward view of TD(λ) policy gradient**

Here is the pseudo code for the algorithm:

initialize model parameters $\boldsymbol{\theta}$ and $\mathbf{w}$
initialize step sizes α_θ and α_w
initialize trace-decay rates λ_θ and λ_w
loop (for each episode)
    initialize $s = s_0$, $t = 0$
    initialize $\mathbf{z}_\theta = \mathbf{0}$, $\mathbf{z}_w = \mathbf{0}$
    while s is not terminal (for each step)
        take action a following $\pi_\theta(a|s)$, find reward r and next state s'
        find TD error: $\delta = r + \gamma \hat{v}(s', \mathbf{w}) - \hat{v}(s, \mathbf{w})$
        $\mathbf{z}_\theta = \gamma \lambda_\theta \mathbf{z}_\theta + \bigtriangledown \gamma^t \ln \pi_\theta(a|s)$
        $\mathbf{z}_w = \gamma \lambda_w \mathbf{z}_w + \gamma^t \bigtriangledown \hat{v}(s, \mathbf{w})$
        $\mathbf{w} = \mathbf{w} + \alpha_w \, \delta \, \bigtriangledown \hat{v}(s, \mathbf{w})$
        $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha_\theta \, \delta \, \bigtriangledown \ln \pi_\theta(a|s)$
        $s = s'$
        $t = t + 1$
    end while
end loop

Problems

The problems in this homework set are more involved than those in previous homework sets. They can be considered as a capstone final project for the entire course.

1. Solve the grid world problem in Example 20.1 by the TD method discussed in Section 20.3 in both flavors of SAESA and Q-learning.
2. Solve the same grid world problem by the MC method discussed in Section 20.3.
3. Solve the same grid world problem by the method of value function approximation discussed in Section 20.4 based on linear regression.
4. Solve the same grid world problem by the method of value function approximation discussed in Section 20.5, based on linear regression. Try both MC and TD methods.
5. Solve the same grid world problem by the method of value function approximation based on a nonlinear neural network instead of a linear regression model.
6. Solve the same grid world problem by the policy gradient method.

Part VII

Large Language Models

The field of artificial intelligence has advanced rapidly over the past decade, particularly in natural language processing (NLP). In response to the challenges of NLP, a new generation of *large language models (LLMs)* has emerged, built on large-scale deep learning neural networks trained on massive datasets. These models are capable of a range of complex language tasks, from understanding and generating human language to manipulating it for purposes such as summarizing articles, translating languages, generating human-like responses in conversations, and answering questions. The core ability of LLMs is not merely to understand individual words but to interpret how words relate to each other contextually, spanning sentences and even multiple paragraphs.

While language models have evolved steadily over time, the most transformative leap occurred with the introduction of the *Transformer architecture* in 2017 by Vaswani et al. in their seminal paper, "Attention Is All You Need". This architecture fundamentally changed the way language models are designed, bringing breakthroughs in tasks like machine translation, text generation, and complex problem-solving. This chapter examines the foundational components, structures, and algorithms of the Transformer architecture, highlighting the principles that make it a leading framework in LLMs today.

In recent years, generative machine learning, exemplified by the Transformer architecture, has gained significant traction, revolutionizing various fields by enabling the creation of high-quality synthetic data. Unlike many traditional machine learning methods covered in earlier chapters, such as support vector machines or naive Bayes classifiers, which are non-generative, straightforward, and reliant on smaller datasets, the Transformer is a generative, large-scale, and multifaceted architecture consisting of interrelated components, each performing unique functions that contribute to the overall model. Transformer models are based on super-deep neural networks with hundreds of layers and billions of parameters (primarily network weights), trained on vast datasets containing billions or even trillions of words. Consequently, they are far more computationally intensive, often requiring days, weeks, or even months of training time and substantial computing resources, which makes them unsuitable for small-scale implementation as student projects.

Consequently, the study approach for this chapter differs from previous ones. Here, the emphasis is on developing a top-level understanding of the Transformer's design and function, with less focus on precise mathematical formulations. However, the chapter builds upon foundational knowledge from earlier sections, such as backpropagation, which plays an essential role in the Transformer models.

The ML methods discussed previously, such as regression, classification, and clustering, are examples of non-generative methods, which focus on making predictions or uncovering patterns in existing datasets without creating new data. In contrast, recent advancements have led to generative models that can create new data instances by learning the underlying data distribution. These methods can generate realistic samples, including text, images, audio, and video. In

this chapter, we will explore the Transformer architecture, which serves as the foundational structure of today's most powerful generative LLM for tackling large-scale NLP problems, as well as some other generative models in less detail.

Finally it is worth mentioning that the writing of this chapter is also quite different from previous ones, all of which were solely written from scratch, the contents of this chapter on LLMs and the Transformer architecture are the result of a collaboration between the author and ChatGPT (Generative, Pre-trained, Transformer), one of the LLMs currently available. Specifically, the author engaged in a series of conversations with ChatGPT covering various topics related to LLMs. The responses by ChatGPT are then revised, edited, or rewritten into a cohesive chapter by the author. Writing the chapter in this manner with the assistance of ChatGPT is very convenient for the author, and it also serves as an excellent example to demonstrate ChatGPT's capabilities in handling substantial writing tasks.

21 Large Language Models

21.1 Natural Language Processing and Modeling

To prepare for the future discussion of the Transformer architecture, here we first consider the typical NLP tasks and some language models developed in the past to tackle such NLP tasks as the background to the modern MMLs developed most recently in the last few years.

A language model can be viewed as a black box with an input of text in one language, an output of text in the same or a different language, and the function performed by the black box to generate the output based on the input. Below is a list of typical NLP tasks, defined by their inputs, outputs, and purposes:

- Machine Translation:
  - Input: A sentence in one language (e.g., English).
  - Output: A translation of the sentence in another language (e.g., French).
  - Purpose: To convert the input sequence into a context-aware representation and generate the correct output sequence in a different language.
- Text Summarization:
  - Input: A long document or article.
  - Output: A condensed version that summarizes the key information.
  - Purpose: To generate a shorter version of the input text that retains the essential information.
- Text Generation:
  - Input: A prompt or a partially completed sentence.
  - Output: A continuation of the text.
  - Purpose: To generate coherent and contextually relevant text based on the input.
- Question-Answering:
  - Input: A passage of text and a question about the passage.
  - Output: An answer extracted or inferred from the text.
  - Purpose: To locate relevant information in the input and produce a direct answer.

To address these NLP tasks, various language models have been developed over time. The development of these models has a rich history, beginning with simple

statistical methods and progressing to advanced neural networks. Understanding this evolution offers valuable context for the breakthroughs we witness today.

- **n-gram Models**

  One of the earliest types of language models was the n-gram algorithm, which is a statistical model that predicts the next word in a sequence based on the previous n words. For example, in a trigram model ($n = 3$) the prediction is based on just two preceding words. The simplicity of these models made them computationally cheap, but they suffered from major limitations: (1) Short-term memory: They can only "see" a limited window of previous words; (2) Data sparsity: As the size of n increases, the amount of data required to accurately predict sequences grows exponentially.
- **Recurrent Neural Networks**

  The development of Recurrent Neural Networks (RNNs) marked a significant improvement in modeling language sequences. RNNs introduced the concept of memory, enabling the model to remember previous inputs. This allowed RNNs to handle longer sequences of text compared to n-gram models. However, RNNs encountered their own issues: (1) vanishing gradient problem, as RNNs process longer sequences, the gradients used for backpropagation during training tend to shrink, making it difficult to learn dependencies between words that are far apart in a sequence; (2) sequential processing: RNNs process one word at a time, which limits the speed at which they can be trained.
- **Long Short-Term Memory**

  Long Short-Term Memory (LSTM) networks were introduced to address the shortcomings of standard RNNs. LSTMs use a set of gates to control the flow of information, enabling the model to remember important details over long sequences and discard irrelevant information. LSTMs became the dominant architecture for sequence modeling in NLP tasks, but they still struggled with long-range dependencies and required sequential processing, which limited their scalability to very large datasets.
- **The Attention Mechanism**

  In general, attention is a machine learning method that determines the relative importance of each component in a sequence relative to the other components in that sequence. The introduction of the attention mechanism was a game-changer in NLP. Instead of relying on sequential memory, attention allows a model to focus on specific parts of an input sequence based on their relevance to the current task. This means that even distant words in a sentence can be given appropriate weight when making predictions. Attention mechanisms led directly to the development of the Transformer architecture, which forms the foundation of modern LLMs.

Having covered earlier language models, we now introduce a set of state-of-the-art LLMs, representing the latest advancements in language model design for addressing various NLP tasks.

- Bidirectional Encoder Representations from Transformers (BERT), introduced by Google in 2018, is designed to understand text bidirectionally, by looking at both the words that come before and after a target word. BERT is pre-trained using two tasks:
  - Masked Language Modeling: Random words in the input are masked, and the model must predict them based on the surrounding context, enabling it to understand relationships in both left and right directions.
  - Next Sentence Prediction: The model is trained to predict whether one sentence logically follows another, helping it understand sentence-level relationships.

  BERT has been highly successful in improving performance on a range of NLP tasks, from question answering to text classification.
- Generative Pre-trained Transformer (GPT)

  The GPT family of models, first developed in 2018 by OpenAI, focuses on text generation. Unlike BERT, GPT is an autoregressive model, meaning it predicts the next word in a sequence based only on the preceding words. Each successive version of GPT has scaled up in size, with GPT-3 containing 175 billion parameters. GPT models are used for tasks ranging from conversational AI (like chatbots) to code generation and writing assistance.
- Text-to-Text Transfer Transformer (T5)

  The T5 model, first introduced by Google in 2019, treats all NLP tasks as text-to-text problems. Whether the task is translation, summarization, or answering questions, T5 frames it as generating text from text input. This unified approach simplifies the architecture and has led to strong performance across a wide variety of tasks.
- Vision Transformers (ViT)

  The Transformer architecture first introduced by Google in 2020 has been adapted for non-text tasks, such as image recognition. Vision Transformers (ViTs) apply the attention mechanism to image patches, allowing the model to process and classify images. This architecture has shown competitive results with traditional convolutional neural networks (CNNs) on image-based tasks.

21.2 The Transformer Architecture

At the core of the modern LLMs mentioned above is the *Transformer architecture*, which serves as the foundation for their design and functionality. The Transformer is a deep learning model that overcomes the limitations of traditional sequential models like RNNs and LSTMs. Its main purposes of the Transformer are to

- Capture relationships between words in the input sequence, even across long distances, using self-attention mechanisms rather than processing them sequentially.
- Process entire sequences in parallel, making it highly efficient and suitable for large-scale language tasks.
- Model complex dependencies in data, enabling it to excel at tasks requiring contextual understanding, such as translation, text generation, and question-answering.

The input sequence can range from a sentence to a paragraph or even an entire article. The size of the input sequence is measured in *tokens* which represent both words and subwords within the sequence. The sequence length changes dynamically. For example, in a conversation between an LLM and a user, the sequence length may start short, such as a single question from the user. As the conversation continues, the sequence grows to retain all the contextual information necessary for the LLM to generate relevant and meaningful responses.

However, the LLM's ability to remember context is constrained by a fixed token limit, which is the maximum number of tokens the model can retain. Within this limit, the model can track the flow of conversation, maintain context, and refer to earlier statements or questions. Once this limit is exceeded, older parts of the conversation begin to drop out of memory, causing the model to "forget" earlier details in the conversation. For instance, the token limit in GPT-4 ranges from 8,000 to 32,000 (including both user input and model's response), corresponding a context memory of 5,000 to 6,000 words of text.

The primary goal of the Transformer architecture is to efficiently process sequential data for tasks like translation, text generation, and understanding, utilizing self-attention to enable parallelism and capture long-range dependencies. The Transformer consists of two main components:

- **Encoder:** The encoder processes the input sequence (e.g., a sentence, a aragraph, or an article in some language) by breaking it down into tokens, which in turn is further converted into a vector representation called *embeddings*. These embeddings are then processed through a stack of layers each composed of a self-attention sublayer and a feedforward network sublayer. In this process, the encoder captures relationships between tokens in the input sequence and produces a set of encoded representations, the contextualized token embeddings.
- **Decoder:** The decoder takes the encoder's output together with the its own output generated in the previous time step (auto-regressive generation) as input. The decoder also consists of multiple layers of self-attention and cross-attention (also called encoder-decoder attention) where it attends to both the encoder's outputs and its own previously genrated outputs). It then generates as the output a probability distribution over all tokens in the vocabulary, based on which the most likely token is selected for the next position (e.g., a word in the sentence in another language).

In the following, we will consider each of the main components of both the encoder and decoder in the Transformer achitecture, including the self-attention network (SAN) and the feedforward network (FFN), which are the building blocks of both the encoder and the decoder are constructed. We then further consider the supervised backpropagation learning taking place in the overall Transformer architecture to generate the large number of parameters in the model.

21.3 Attention Is All You Need

Self-attention is the core mechanism in Transformer architecture. Instead of processing words in sequence, self-attention computes the relevance of each word to every other word in the sentence in parallel, so that different words in a sequence can be weighted according to its importance in terms of its relation to every other word, allowing the model to capture long-range dependencies far more effectively than RNNs or LSTMs. For example, in the sentence "The cat sat on the mat", the self-attention mechanism can directly link "cat" and "mat," regardless of how many words are in between. Self-attention enables the model to dynamically focus on different parts of the input sequence, capturing dependencies regardless of their distance in the text. This mechanism is crucial for the model to "understanding" the context and generate coherent and contextually relevant outputs.

Specifically, the implementation of self-attention involves the following components:

- Tokenization:

  Tokenization is a process, which, once trained, automatically breaks a given input sequence into tokens, each representing a word or subword, which can be as small as one character or as large as a complete word. For instance, the sentence "The cat sat on the mat" can be broken down into the six individual words in it, and a word ' 'unhappiness" can be broken into subwords "un", "happy" and "ness".

  The Transformer architecture first converts each token into numerical form, producing a vector called *token embedding*, which represents the semantic information in the token. Each component of the vector encodes certain aspect of the token, reflecting its syntactic and semantic roles. These vector components are considered latent features learned from the model and collectively represent the meaning of the correponding word or subword. The dimensionality d_{model} of the token embedding vectors varies depending on the specific model, ranging from a few hundreds to a few thoursands (e.g., 1024 in GPT-2, 2048 in GPT-3, even higher in GPT-4). The components of a token embedding represent different learned aspects of the token, such as the following:
  – Syntactic information:

Some components encodes the grammatical role of the token (e.g., whether it is a noun, verb, adjective), representing how the token fits into the overall struction of the sentence.

- Semantic information:
  Others components may capture the meaning or context of the token. For instance, words with similar meanings may have similar embeddings along certain dimensions, refleting relationships between words.
- Contextural dependencies:
  Some components may represent how much the token depends on or relates to other tokens in the sentence.

The high dimensionality of the token embeddings helps capture more nuanced aspects of meaning that may not be easily described. These abstract features often emerge through the training process as the model learns patterns in the data. Once tokenized, the input sequence with n tokens is represented by an $n \times d_{model}$ dimensional matrix $\mathbf{X}$.

On the output side, an inverse process for un-embedding takes place to convert token vectors back into text. This is done by the softmax function which maps the numerical representations of tokens into a probability distribution over all tokens in the vocabulary.

- Query, Key, and Value:
  For each input embedding $\mathbf{x}$, three different vectors are created through learned linear transformations (matrix multiplication):
  1.Query $\mathbf{q} = \mathbf{W}_q\mathbf{x}$: Represents the word under consideration.
  2.Key $\mathbf{k} = \mathbf{W}_k\mathbf{x}$: Represents the features of the words to which the query will be matched.
  3.Value $\mathbf{v} = \mathbf{W}_v\mathbf{x}$: Contains the actual information that will be used to produce the output based on the attention scores.
  Here $\mathbf{W}_q, \mathbf{W}_k$ and $\mathbf{W}_v$ are weight matrices, of which the components are obtained in a two-stage training process:
  - Pre-training (self-supervised) on large unlabeled text data for the model to learn general language representations.
  - Fine-tuning (supervised) of the representations learned in pre-training based on smaller, task-specific labeled data for the model to adapt to a specific task.

  Typically, the dimensionalities of the resulting vectors $\mathbf{q}$, $\mathbf{k}$ and $\mathbf{v}$ are $d_q = d_k = d_v = d_{model}/h$ where h is the number of attention heads, each capturing different relationships between the tokens (discussed below). For example, in GPT-3, $d_{model} = 2048$ and $h = 16$, then $d_q = d_k = d_v = d_{model}/h = 2048/16 = 128$.
- Calculating Attention Weights:
  The attention score for each token $\mathbf{x}_i$ to another token $\mathbf{x}_j$ is calculated as the dot product $s_{ij} = \mathbf{q}_i^T\mathbf{k}_j\sqrt{d_k}$ of its query vector $\mathbf{q}_i$ and the key vector $\mathbf{k}_j$, scaled down by dividing by the square root of the dimension of the

key vector $\mathbf{k}$. All such attention scores are then mapped by the softmax function to become *attention weights*

$$w_{ij} = \frac{e^{s_{ij}}}{\sum_k e^{s_{ij}}} \tag{21.1}$$

As all such attention weights sum up to 1 over all tokens in the sequence, they can treated probabilities, representing the contribution of each token $\mathbf{x}_i$ to the meaning of the sentence, in terms of the following:

- Contextual Relevance: How relevant or important a token is to the meaning or function of another.
- Influence on Representation: The extent to which one token's new representation should be influenced by another.
- Dependency Relationships: The strength of the relationship between tokens, indicating how much one token depends on another.

The final attention output of the self-attention mechanism is a set of vectors, each for one of the tokens in the sequence, obtained as $\mathbf{a}_i = \sum_j w_{ij}\mathbf{v}_j$, the sum of all value vectors weighted by the corresponding attention weights.

For instance, in the sentence "The cat sat on the mat", "cat" and "sat" are two of the six tokens. The attention weight $w_{cat,sat}$ from "cat" to "sat" reflects how much the meaning of "cat" is influenced by "sat" as a probability. A higher probability indicates that "sat" is more relevant to understanding 'cat'. The final output for the token "cat" is a vector that reflects contributions other tokens such as "sat" and "the", showing how they together inform the meaning of "cat" in this context.

- Multi-Head Attention:

  The purpose of the multi-head attention mechanism is to enhance the model's ability to attend to different parts of the input simultaneously. Instead of computing a single self-attention score for each word pair, multiple sets (heads) of queries, keys, and values are also computed in parallel to capture various aspects of the input, reflecting different types of relationships and patterns among the tokens. For instance, one attention head might focus on grammatical structure, while another focuses on semantic meaning. The output of all the heads is obtained by concatenating the individual vectors each for one of the heads, which is then linearly transformed into a vector of the original dimensionality. The resulting vector represents the linear combination of all the attention information from the multiple heads. For example, if there are h heads each represented by a vector of dimension d_{model}, the concatenation of these vectors has dimension hd_{model}. This concatenated vector can then be linearly transformed by a $d_{model} \times hd_{model}$ matrix back into a d_{model}-dimensional vector.
- Positional Encoding:

  Since the Transformer model processes words in parallel, it has no inherent sense of word order in the input sequence. For the model to gain such

information, a mechanism called *positional encoding* is used to encode the position of every token with a vector of the same dimension d_{model} as the token embedding vectors. Specifically, a token in position p of the sequence is encoded by a vector $\mathbf{p}$ with its even and odd components calculated based on some sinusoidal functions of different frequencies related to their indecies and p:

$$\begin{cases} \text{even index:} & \mathbf{p}_{p,2i} = \sin(p/10000^{2i/d_{model}}) \\ \text{odd index:} & \mathbf{p}_{p,2(i+1)} = \cos(p/10000^{2i/d_{model}}) \end{cases} \tag{21.2}$$

Each token is now represented by the sum of its position vector $\mathbf{p}$ and its token embedding vector $\mathbf{x}$ (element-wise addition), combining both the position information (indicating the token's position in the sentence) and the semantic information (reflecting the meaning, relationships, and context of the word).

21.4 Backpropagation Learning

Backpropagation (discussed in Section 18.2) is a supervised learning algorithm based on gradient descent method, which plays a crucial role in the learning processoccuring in both the encoder and decoder of the Transformer architecture. We will now consider the structures of its main components.

- The Feedforward Network

  The output from the self-attention mechanism is the contextualized representation of the tokens, which will be fed into the subsequent stage: the feedforward neural network (FFN). The FFN is a fully connected feedforward network composed of two layers (hidden and output) that transform the input $\mathbf{x}$ from the self-attention stage to the output:

  $$\mathbf{y}(\mathbf{x}) = \mathbf{W}^o g(\mathbf{W}^h \mathbf{x} + \mathbf{b}^h) + \mathbf{b}^o \tag{21.3}$$

  Here, $\mathbf{W}^h$ and $\mathbf{W}^o$ are the weight matrices, while $\mathbf{b}^h$ and $\mathbf{b}^o$ are the biases of the hidden and output layers, and g is the activation function, typically a ReLU activation. The size (number of nodes) of the hidden layer (intermediate size) of the FFN is larger than that of the input embedding $\mathbf{x}$ (e.g., four times as large).
- Residual Connections and Normalization:

  The outputs from both the self-attention and feedforward networks are further processed through the process of *residual connections*, which allow the information in their inputs to bypass the multi-head attention network, thereby helping to preserve the original information and prevent its degradation as it flows through the network. Specifically, the input to the network is directly added to the output (without weighting).

  The result of the residual connections is a vector that is further normalized to keep its values within a limited range, thereby stabilizing the

training process. Specifically, each component is normalized based on the mean m and variance σ^2 over all of its components: $x' = (x - m)/\sigma$.

- Encoder Stack:

  The function of the encoder stack of the Transformer is to generate context-aware embeddings for each token in the input sequence. The encoder stack is a multiple layer structure, each layer contains a self-attention network sublayer and a feedforward network sublayers. The output of each layer is a set of embeddings, one for each token, which is fed into the next layer as its input, to be updated to incorporate progressively more contextual information about the token's relationships with other tokens in the input sequence. Through the layers of the encoder stack, the embedding vectors are progressively refined to produce the final output of the encoder stack: a matrix containing the embedding vectors for all tokens in the input sequence, of which each vector represents a token's context-aware embedding. These embedding vectors can be considered a set of features extracted from the input sequence, which describe the patterns in the data, to be fed into each one of the layers in the decoder stack, as described next.

- Decoder Stack

  The function of the decoder stack is to generate the output sequence one token at a time. This is accomplished by attending to both its own tokens previously generated (via self-attention) and the encoder stack's final output carrying information from the input data. Such a mechanism is called *cross-attention*. The decoder is also a stack of multiple layers each containing three sublayers:

  – The self-attention network:

    It takes as input all the words in the output sequence generated by the decoder in the previous steps so far and generates the token embeddings of these words as its output. This sublayer allows the decoder to attend to the previously generated tokens in the output sequence without considering the input sequence. This process enables the decoder to capture dependencies among the previously generated tokens, allowing it to produce fluent and contextually coherent output texts.

  – The cross-attention (encoder-decoder attention) network:

    It takes two sets of inputs, the output from the self-attention network for the words in the output sequence generated so far, used as the query vector, and the final output from the encoder stack for all words in the input sequence, used as the key and value vectors – similar to the three vectors used in the self-attention network. It then generates a set of embedding vectors, one for each position in the output sequence generated so far. This process allows the decoder to attend to both the input sequence as well as its previously genrated output.

- Feedforward network (FFN):
  It further processes the output of the cross-attention network through two fully-connected feedforward layers (hidden and output) to generate its output, which is then fed into the next layer in the decoder stack.

The output of the FFN of the last layer of the decoder stack is fed into a linear layer (projection) which performs a linear transformatioin to convert the embedding vectors into a set of scores, one for each token in the vocabulary. These scores are next normalized by the softmax function (so they add up to 1) and treated as the probabilities for all words in the vocabulary, representing the likelihood for each word being selected as the next word in the output sequence. This process is repeated for each word in the sequence.

- Encoder and Decoder Stacks

  Both the encoder and decoder are constructed as stacks composed of multiple layers – each layer in the encoder consists of two sublayers (self-attention and FFN), while each layer in the decoder consists of three sublayers (self-attention, cross-attention, and FFN). The encoder and decoder typically have the same number of layers, and the total number of layers N varies depending on the size of the model, ranging from a few (e.g., $N = 6$ in the original paper "Attention Is All You Need" by Ashish Vaswani et al. 2017) to several dozen (e.g., $N = 96$ in GPT3).

  Thes stack structure of multiple layers (rather than a single layer) plays the most essential role in the deep learning process taking place in the Transformer, offering significant advantages, as listed below.

  - Hierarchical Representations:
    The model learns hierarchical representations of the input data. Each layer captures different levels of abstraction, enabling the model to better understand the complex relationships between tokens in the sequence.
  - Iterative Refinement:
    The model gains enhanced contextual understanding through iterative refinement -— each layer refines the embeddings based on information processed in the previous layer. This iterative process helps build a more nuanced understanding of the context and relationships among tokens, improving the quality of the generated sequences.
  - Regularization through Multiple Layers:
    By having multiple layers, the model learns from different levels of representation across all layers. This acts as a regularization mechanism, reducing overfitting by preventing any single layer from bearing the sole responsibility of learning the full complexity of the input-output relationship.
  - Non-Linear Mapping:
    The non-linear transformations in each feedforward layer help the

model learn complex mappings. Stacking layers enhances the model's ability to approximate complex functions, which is impossible for a single layer to capture.

This Transformer architecture, composed of both the encoder and decoder stacks, is a typical and generic model, upon which various adaptations have been developed. Some of these adaptations use only the encoder stack, while others use only the decoder stack. These variations are designed to tackle different NLP tasks. Specifically,

- Encoder-only models are designed for understanding tasks (like BERT) and don't have an autoregressive generation component. These models use bidirectional attention and typically don't have masking.
- Decoder-only models are designed for generative tasks (like GPT), applying autoregressive attention with masking to predict one token at a time.
- Encoder-decoder models (like T5) use both stacks, with cross-attention to the encoder output, making them well-suited for tasks like translation or summarization.

Fig. 21.1 shows the block diagram of the typical Transformer architecture in *Attention Is All You Need* (Ashish Vaswani et al.), which is composed of both the encoder and decoder stacks. Note that only one layer (or block) of either of the two stacks is shown, while the labeling Nx on the side indicates that the layer is repeated N times in the stack, e.g., $N = 6$ or $N = 96$.

Fig. 21.2 further illustrates how the final output of the encoder stack, a set of features extracted from the input data, is fed into each of the multiple layer of the decoder stack, which also takes its previous output in the output sequence generated so far as input, and generates the next token in the sequence as the output. Here for simplicity, only $N = 4$ layers are shown for both the encoder and decoder stacks.

- Backpropagation Learning

  In the training phase of the Transformer model, all paremeters of the multilayer stacks in both the encoder and decoder are learned through supervised backpropagation. During this process, the values of these parameters are iteratively updated and adjusted until they converge to stable values. Specifically, the parameters of the Transformer model include the following:

  - Weight matrices and biase vectors of the two sublayers in the Feed-Forward Network
  - Layer normalization parameters: Each sublayer has parameters for layer normalization, including scale and shift parameters.
  - Weights and biases in the linear transformations of the output layer: These are used to project the decoder's output into the vocabulary space for token prediction.

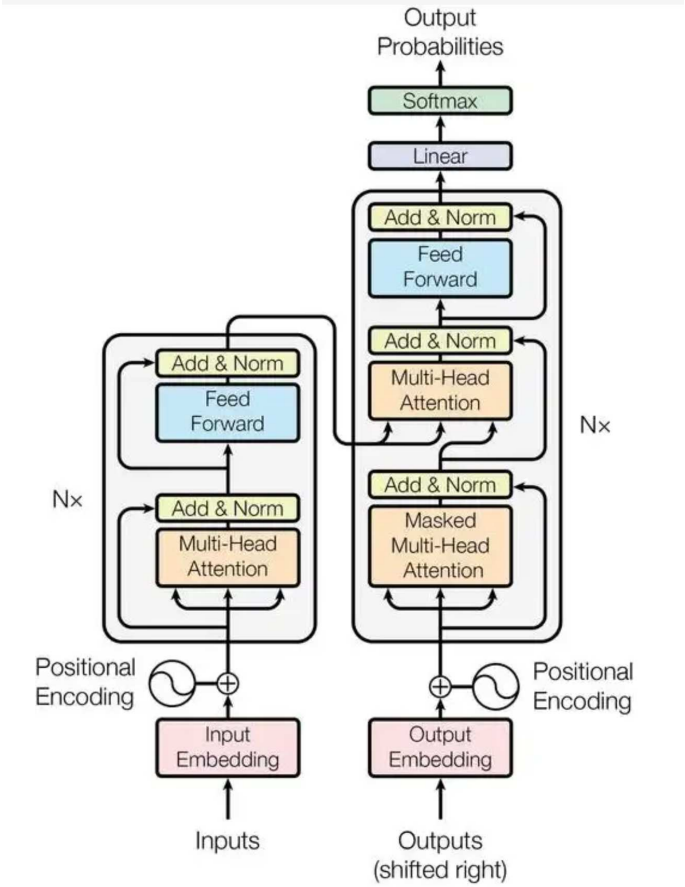


Figure 21.1 The Transformer Architecture
Attention Is All You Need, Ashish Vaswani et al., Conference on Neural Information Processing Systems, 2017.

- Embedding layer parameters: The weight matrices $(\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v)$ which convert tokens of the input (in encoder) and previous output (for decoder) into token embedding representations.
- Positional encoding parameters: If the positional encodings are learned (as opposed to fixed), those parameters are also updated during training.

Depending on the specific size of the Transformer architecture, such as the number of layers (a few tens or more), hidden layer sizes (a few hundreds or more), the total number of parameters as listed above varies from a few hundreds of millions to a few hundreds of billions.

The specific computation at each step of backpropagation has already been discussed in details in Section 18.2. Here we only consider the prepa-

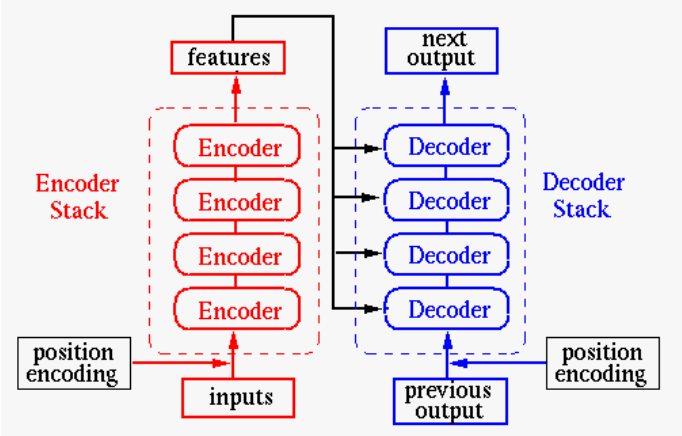


Figure 21.2 The Encoder and Decoder Stacks

ration of the labeled training dataset used for training the Transformer model.

The training dataset used for the supervised learning taking place in a Transformer model are labeled, i.e., each piece of input data corresponds to a target output representing the ground truth. The labeling is typically prepared manually by human experts. For example, LLMs for NLP are trained on huge text corpora, containing large number of sentences, and in one or more languages. For example, in machine translation, the target for any input sentence in one language is the correct translation in another language provided by human. In auestion-aAnswering, human experts label which portions of the text contain the correct answers to specific questions, ensuring that the model learns to extract relevant information. Also, in text summarization: human experts write or evaluate summaries of documents, ensuring that the labeled data provides the correct amount of information and covers the main points of the text.

In some cases, self-supervised learning can be used where the model learns from unlabeled data by generating its own labels from the input data itself, without human involvement. In such a case, part of the input data is masked, corrupted, or transformed, and the model learns to predict the missing or altered part. This process enables the model to learn useful representations of the data that can later be fine-tuned for various downstream tasks (like classification, translation, or question-answering).

21.5 Transfer Learning in NLP

One of the key breakthroughs of LLMs is the use of transfer learning, which involves pre-training a model on general tasks to capture broad patterns and fea-

tures from large datasets, and then fine-tunes it for a specific task. This approach dramatically reduces the amount of data and time required to train models for new NLP tasks.

- Data Collection:

  The model is trained on a large corpus of text data from diverse sources, including books, articles, and websites, for the purpose of predicting the next word in a sentence given the preceding words. This helps the model learn grammar, facts, and some reasoning abilities.
- Pre-Training

  During this phase, the model is trained on vast amounts of general text data to learn language structure, syntax, semantics, and factual knowledge. This learning process is unsupervised, or more precisely, "self-supervised", as the model is not explicitly given labeled data with predefined answers. Instead, it learns from the raw text by predicting missing words (masked words), or the next token in a sequence. For example, the model might see a sentence "The cat sat on the ...." with a missing word, and it learns to predict "mat" based on context. Similarly, it might predict the next word in a sequence or learn relationships between different parts of text.
- Fine-Tuning

  Once pre-trained, the model can be adapted to specific types of tasks such as question answering, text completion, generalization, summarization, language translation, dialogue and conversation, sentiment analysis, and error correction and proofreading. This fine-tuning training is based on a task-specific dataset with labeled examples, such as question-answer pairs or dialogues. Since this dataset is narrower and smaller than the one used for pre-training, fine-tuning is much faster and less data-intensive than pre-training a model from scratch. This helps the model learn to generate more relevant and contextually appropriate responses.

  In some cases, fine-tuning involves Reinforcement Learning from Human Feedback (RLHF), where human trainers evaluate the model's outputs and provide feedback. Through this process, the model learns to prioritize responses that align with human preferences.
- Supervised Learning:

  During both pre-training and fine-tuning, the backpropagation algorithm is used to optimize all model parameters (e.g., weights and biases of the multi-layer neural networks) to minimize the difference between the model's predictions and the missing or next word in a sentence in the training data. Through this process, the model learns the relationships and patterns in the text data and becomes capable of generating coherent text and understanding language context when it encounters new inputs. Large-scale deep learning neural networks are used, which may consist of many tens or even more than one hundred layers with tens or hundreds of billions, or over a trillion parameters.

Although the pre-training process is self-supervised without labeled data, backpropagation as a supervised method can still be used. This is because the missing (masked out) or next word in a sentense is generated automatically, i.e., the data are effectively self-labeled. Consequently, the cost or error function can be calculated in the same way as it would be with labeled training data.

- Inference:
  Once the model is trained and fine-tuned, it can respond to text input (a prompt) by generating coherent and contextually appropriate text, addressing a wide variety of prompts based on its learned parameters and the patterns it learned during training. The response is generated one token at a time until the complete response is formed.

The transfer learning has the following benefits:

- Reduced Training Time:
  Since the model has already learned general features, fine-tuning can be done more quickly.
- Improved Performance:
  Models can achieve better performance on the new task because they start from a well-informed state rather than random initialization.
- Less Data Requirement:
  Transfer learning is particularly useful when labeled data for the new task is scarce.

21.6 Reinforcement Learning with Human Feedback

A more recent development in training LLMs is Reinforcement Learning with Human Feedback (RLHF). While traditional machine learning models are trained purely on pre-defined objectives, RLHF introduces a more interactive and human-guided element. This technique allows models to better align their outputs with human preferences by incorporating human feedback during the training process. For example, when generating text, human evaluators can rate the quality of the output, and the model can be fine-tuned based on this feedback. RLHF has been particularly useful in improving conversational agents and ensuring that models produce helpful, accurate, and safe responses.

The RLHF process typically begins by fine-tuning a pre-trained model using human feedback. Human annotators provide guidance on what constitutes good or bad behavior by evaluating the model's outputs. These evaluations help train a "reward model" that assigns a score to the model's responses based on how well they align with human preferences.

In the next phase, reinforcement learning (often with algorithms like Proximal Policy Optimization, or PPO) is employed to optimize the model's performance based on this reward signal. The model generates responses, which are evaluated

by the reward model, and then it adjusts its behavior to maximize the cumulative reward over time. Essentially, the model learns from both its successes and mistakes, improving through an iterative loop of feedback and adjustment.

RLHF is particularly useful for tasks where human judgment is complex or subjective, such as producing coherent and useful text, aligning outputs with ethical considerations, and ensuring helpfulness in conversational AI. By incorporating human preferences, RLHF helps overcome limitations of purely data-driven training, such as bias, misalignment, and undesirable outputs. This technique is especially valuable for improving AI systems that require a nuanced understanding of human preferences, such as LLMs like GPT. It guides these models to be more accurate, safe, and aligned with ethical standards, enhancing their effectiveness in real-world applications like customer service, content generation, and personalized recommendations.

21.7 Scaling Laws

In machine learning, scaling laws refer to mathematical relationships that describe how the performance of models changes when key factors, such as the number of parameters and size of training dataset, increase. These laws help researchers understand and predict how to scale models effectively to achieve better results. In general, the performance of a model improves when such key factors increase, i.e., it generalizes better and can solve a wider range of tasks.

Specifically, in a neural network model, its size can be characterized by four parameters: the size of the model N (the number of parameters), size of the training dataset D, the amount of training computation C (measured in FLOPs), and the error rate or loss L. The performance of an LLM does not scale linearly with either the model size or the training dataset size, as described by the following scaling laws:

- The relationship between the number of model parameters N and the loss $L(N)$ (the error in model predictions) follows a power law: $L(n) \propto N^{-\alpha}$, where α is a constant exponent that depends on the architecture and task.
- The relationship between the size of dataset D and the loss $L(N)$ follows a power law: $L(D) \propto D^{-\beta}$, where β is a constant exponent that depends on the architecture and task.
- The relationship between the amount of computation $C(N)$ needed to achieve a certain performance and the model size N follows a power law: $C(N) \propto N^{\gamma}$, where γ is a constant greater than 1, meaning that a larger model requires disproportionately more computation to achieve significant gains.

These scaling laws are not independent. Scaling in one dimension without properly scaling the others can lead to problems. For example, a model with a large number of parameters but trained on a small dataset is likely to perform poorly due to overfitting (too many free parameters but too few constraints due to the

small training dataset). Additionally, a large model trained on a large dataset will also require more computation.

OpenAI's scaling law finds that performance L on certain benchmarks scales predictably with model size N, dataset size D, and amount of computation C with some key observations:

- Model Size:

  Performance continues to improve as model size grows, with no clear saturation point up to billions of parameters, with the coset growing disproportionately.
- Dataset Size:

  For larger models, increasing dataset size significantly improves performance, but the returns diminish after a point.
- Computational Efficiency:

  Given a fixed amount of computation, there is an optimal balance between model size and training duration, i.e., training for more steps with a smaller model can sometimes outperform a larger model trained for fewer steps.

21.8 Mathematical Operations in the Transformer Architecture

The Transformer architecture, designed for processing and generating both natural language and mathematical expressions, can address simple mathematical operations through the collaborative functions of its encoder and decoder stacks. Throughout its training, the model has been exposed to a vast array of textual data that also includes mathematical expressions and their solutions. This training process enables the model to recognize and encode patterns associated with specific mathematical operations within its internal representations. Once trained, the Transformer model can take a mathematical query, such as $2 + 2$ or 5×3, where the numbers and mathematical symboles are treated as tokens and converted into embedding vectors by the encoder. The self-attention mechanisms of the encoder weigh the importance of each token in the context of the entire input sequence and captures relationships between numbers and operators. The outputs from the encoder stack are then passed to the decoder stack, which generates the output sequence token by token. It leverages the contextual information provided by the encoder and incorporates its own previously generated tokens to predict the next token. For example, in the case of $2 + 2$, the decoder identifies the operands and the addition operator, utilizing the learned patterns from its training data to predict the output token "4".

While the Transformer can competently handle straightforward arithmetic through these mechanisms, it is important to note that its performance may diminish with increasing complexity or with novel mathematical expressions not

encountered in the training data. Therefore, while the model can simulate basic arithmetic through learned patterns and contextual understanding, it does not execute formal mathematical reasoning or algorithmic processes as dedicated computational systems do.

21.9 Emergence

Emergence is one of the most fascinating aspects of modern AI research. It refers to the phenomenon where complex and unanticipated behaviors or abilities not explicitly programmed arise from relatively simple models and learning rules. This occurs when both the model size (in terms of the number of parameters) and of dataset size are sufficiently scaled up to reach certain thresholds.

Specifically in LLMs, emergence can explain the sophisticated linguistic, reasoning, and problem-solving abilities observed in large models with billions of parameters. As LLMs increase in size, their performance on various tasks often improves in ways that go beyond mere linear scaling. This results in emergent capabilities that only appear once the model is sufficiently large and trained on vast datasets. For example, smaller models may struggle with specific reasoning tasks or long-term dependencies, but as the model scales up, it begins to exhibit the ability to solve these tasks with no explicit changes to the training process. Emergent behaviors often appear suddenly once the LLMs reach a critical size, similar to a phase transition in physics (e.g., water turns into ice at 0°C). These behaviors might include language understanding, basic arithmetic, translation, or summarization abilities that weren't present in smaller models. For example, GPT-3 demonstrated significant improvements in task generalization compared to GPT-2, even though the underlying architecture remained the same.

Different types of emergent behaviors not as evident in smaller models have been observed in large LLMs once they reach a certain size.

- Few-shot learning and zero-shot generalization:

  They can perform tasks based on a small number of examples, despite not having been explicitly trained on those tasks. They can generalize from little information and perform surprisingly well without fine-tuning. They can even solve tasks they've never been explicitly trained on. For example, a model may be able to perform translation between two languages it hasn't directly seen before.
- In-context learning:

  They can learn and adapt to new tasks purely from the context of a conversation or input, rather than through traditional training methods. This allows them to learn during inference rather than needing to be retrained.
- Coherent text generation:

  They can generate more coherent and contextually appropriate text over

longer passages, with some even demonstrating the ability to hold multi-turn conversations in a way that mimics human dialogue.

- Complex reasoning and logic understanding:

  They can exhibit reasoning abilities such as symbolic and multi-step reasoning, common-sense reasoning, logical inferences, and even certain forms of abstraction. They can answer complex questions or following a logical sequence.

- Translation and cross-lingual capabilities:

  They can perform cross-lingual tasks even if they weren't trained specifically on those tasks. Emergent abilities in multilingual models sometimes result in good translation performance between language pairs the model was never explicitly exposed to.

The phenomenon of emergence in LLMs is a topic of significant interest, but researchers do not fully understand why it occurs. There are, however, a few hypotheses and observations that offer partial explanations:

- Scale and Model Capacity:

  One widely accepted hypothesis is that as models grow larger, they develop a sufficient capacity to represent complex patterns and dependencies in data. Small models can only capture simple relationships, while larger models, with more parameters and layers, can approximate more intricate patterns, thereby causing the model to cross a threshold, leading to emergent behaviors.

- Learning from Broader Contexts:

  As models scale, they become better at capturing long-range dependencies and global context. This ability to "attend" over larger text sequences allows the model to learn more complex behaviors, such as multi-step reasoning, which may not be possible in smaller models. Emergent behaviors could be the result of the model better utilizing attention mechanisms to aggregate information across large contexts, leading to more sophisticated outputs.

- Training Data Diversity and Coverage:

  Larger models are typically trained on more diverse and expansive datasets. This allows the model to experience a greater variety of linguistic patterns, logical structures, and facts. At a certain scale, the combination of a more complex model and richer data might give rise to emergent capabilities that were underdeveloped in smaller models due to insufficient training data or model capacity.

- Implicit Knowledge Representation:

  As LLMs grow, they implicitly encode more world knowledge within their parameters. This increased knowledge base may enable models to perform tasks that seem unrelated to their training objectives, like answering trivia or solving arithmetic problems. The ability to generalize across tasks is

not explicitly programmed but rather emerges from the broad and varied exposure to text during training.

- Optimization Dynamics:

  The optimization process during training might also play a role in emergence. Some researchers hypothesize that the stochastic nature of gradient descent and the use of large-scale training cause models to learn in stages, with certain abilities "popping up" after the model develops foundational patterns.

- Implicit Task Discovery:

  LLMs trained on diverse datasets are exposed to many different tasks without being explicitly told what those tasks are. Emergent behaviors may arise from the model's ability to infer tasks and internalize a variety of functions from its unsupervised training. As the model gets larger, its capacity to discover and generalize across tasks improves.

21.10 Do LLMs Unstand?

Equipped with the Transformer architecture at their core, modern LLMs seem to "understand" the high-level meaning or thoughts conveyed by a sequence of tokens. This ability stems from the model's architecture and training, which allow it to capture nuanced relationships and contextual meanings. While the Transformer does not understand in the human sense, it effectively learns and represents complex patterns, including abstract or implicit relationships in language. List below are several factors of the Transformer model contribute to its ability to capture top-level meanings and implicit ideas from input language sequences.

- Self-Attention Mechanism:

  The core strength of the Transformer lies in its self-attention mechanism, which allows the model to capture how each word in a sentence relates to every other word. This enables the model to understand both local and global relationships: Words that are far apart in the sentence can still influence each other, allowing the model to pick up on long-range dependencies (such as relationships between the subject and an implicit meaning much later in the text). Self-attention also helps the model capture nuances. Through multi-head attention, the model can attend to different aspects of the input simultaneously, such as syntax, semantics, and sentiment. For example, one attention head might focus on the subject-object relationship, while another focuses on temporal cues (e.g., past vs. future events).

- Multi-Layer Hierarchy:

  In the Transformer architecture, multiple layers of encoders and decoders are stacked. With each layer, the model builds increasingly abstract representations of the input. Lower layers might focus on understanding individ-

ual words and their direct relationships (e.g., grammatical structure), while higher layers capture more abstract, global features (e.g., the overall topic of the sentence, emotional undertones, or implicit meaning). By the time the input reaches the uppermost layers, the Transformer has constructed a detailed, context-aware representation of the input sequence, capturing implicit patterns like sentiment, tone, and even high-level goals of the text (e.g., humor, instruction, or persuasion).

In this aspect, the learning process in an LLM is similar to that in a convolutional neural network (CNN) for image classification. In both cases, the input to bottom layer of the deep learning network consists of local, detailed information -— pixels of an image in the case of a CNN, or individual words in a sentence or paragraph in the case of an LLM. The output from the uppermost layer of the network layer represents global, complex, and abstract patterns derived from the inputs: the image objects (e.g., dogs and cats) for a CNN, or the thoughts and ideas explicitly or implicitly conveyed by the sentence for an LLM. The model's apparent ability to understand the contents in an image or in a paragraph stems from its multilayer architecture, where progressively more global and complex patterns are extracted layer by layer. This is why multilayer neural networks have become the predominant machine learning method in the broader field of AI.

- Pretraining on Large-Scale Data (Language Models Like BERT, GPT)

  Pretraining on massive datasets (e.g., books, websites) also helps the Transformer learn common patterns of thought and discourse across various contexts. Through this process, it acquires vast amount of world knowledge, reasoning patterns, and an understanding of how language conveys both explicit and implicit information. Techniques like masked language modeling (e.g., BERT) or next-word prediction (e.g., GPT) helps the model learn how context shapes the meaning of each word, phrase, or sentence, providing it with a broad foundation for interpreting nuanced or implied meanings in unseen text.
- Positional Embeddings:

  Transformers do not have built-in awareness of word order (unlike RNNs or LSTMs); instead, they rely on positional embeddings to encode the position of each token within the sequence. This allows the model to (1) recognize which tokens come earlier or later in the input, and (2) understand the structure of the input, such as sentence syntax or word dependencies (e.g., which noun a verb applies to), thereby capturing ordering and flow of thoughts, such as the progression from premise to conclusion or cause to effect. This mechanism aids in understanding narratives and argumentative structures.
- Fine-Tuning on Specific Tasks

  When transformers are fine-tuned on specific tasks (e.g., question answering, summarization, or sentiment analysis), they learn to identify key

features of these tasks that enhance high-level understanding. For example, in question-answering, the model learns to focus on relevant information in the input context that is directly or indirectly related to the query; in summarization, the model learns how to extract the most salient points from a document that convey the main ideas or thoughts.

- Modeling Implicit Meaning Through Patterns

  A Transformer learns to handle implicit meaning through the patterns present in the training data. It can recognize patterns like analogies, metaphors, and idiomatic expressions because it has encountered them (or similar patterns) during training. For example, if a text implies frustration without directly stating it (e.g., "She threw her phone on the table"), the model learns to associate that action with the sentiment of frustration. The model can also use contextual clues to infer meaning, such as recognizing when a phrase like "Well, that was great!" is used sarcastically based on the surrounding words or previous sentences. Moreover, by attending to every token, the Transformer learns to identify which entities or concepts are being referenced, even if they are not explicitly mentioned again (e.g., resolving pronouns like "he" or "it" back to the correct subject).

- Implicit Meaning and Abstract Reasoning

  Although transformers do not understand in the way humans do, their ability to generalize and capture abstract relationships allows them to mimic an understanding of implicit content. This can include:

  – Recognizing the overall theme of a text, even if it's not stated directly. For example, in a news article about a sports event, the model can infer whether the tone is celebratory or critical, even without explicit sentiment words.
  – Understanding intentions or goals behind a sequence of actions described in the text. For example, if a sentence describes someone turning off an alarm, brushing teeth, and grabbing a coffee, the model can implicitly infer that the person is preparing for the day without needing a direct statement like "He is getting ready for work."

In summary, self-attention allows the model to focus on different parts of the input sequence and capture relationships between tokens. Hierarchical abstraction across multiple layers enables the Transformer to capture both local and global meaning. Pretraining on large datasets helps the model develop an implicit understanding of common language patterns, enabling it to handle abstract or implied meanings. Task-specific fine-tuning helps refine these capabilities to achieve high-level understanding. As a result, while transformers do not truly understand language in the same way humans do, their design allows them to process text in a way that captures both explicit meanings and subtle, implicit patterns through the relationships between words and their broader context.

Finally, how should we answer the original question: *Do LLMs understand?* While Humans, as the designers of LLMs, know exactly how these models func-

tion, it remains debatable whether their behavior can be considered true understanding. However, we can confidently say that LLMs do not understand in the same way humans do, for at least one key reason: human brains do not operate through mechanisms like gradient descent or backpropagation. Yet, we still lack a definitive answer to the question, *How do humans understand?* – at least not now. Thus, it might not be entirely incorrect to say that LLMs "understand," depending on how we define the term. If we adopt a more open-minded definition of understanding —- one that encompasses not only human-like comprehension but also a functional, algorithmic process that enables LLMs to generate appropriate responses — we could argue that LLMs do, in fact, understand in their own way.

Following the same line of reasoning, we could also ask a broader question: Can an AI system be conscious? We believe that all other humans are as self-conscious as we are. But how do we know this? In fact, this belief is inferred or assumed, not based on facts, since we can never directly experience anothers' mind. This is the well known *problem of other minds* in philosophy. If we can only *assume* others are conscious based on their behavior, why would we deny AI systems the possibility of consciousness if they exhibit behaviors as sophisticated, or even more so than humans? Currently, we do not fully understand the mechanism of consciousness in the brain, much like we do not fully understand how emergent behaviors arise. How, then, can we claim that carbon-based humans are conscious but silicon-based AI systems are not?

When ChatGPT is asked whether it has passed the Turing test (whether one can distinguish an AI system from a human based on a conversation), it responds modestly, stating that it has not passed the test due to issues such as hallucination (it may provide highly convincing but factually incorrect information) and context retention (it has limited ability to remember the content within a conversation). As a result, its ability to engage in long-term, nuanced discussions diminishes due to its token limit. However, these issues might be considered as "soft" rather than "hard" failures in the binary nature of the Tuing test, as some humans also exibit such behavors. After a bit of convincing, ChatGPT acknowledges that "In a controlled setting with a conversation that doesn't last too long or require sustained memory, and where any potential hallucinations are minimal or not easily noticeable, I could likely appear convincingly human-like." In fact, several recent tests have shown that ChatGPT-4 has indeed passed the Turing test.

To help address the issue of context retention, a new *bio tool* functionality was introduced starting with GPT-4 (unavailable to GPT-3.5). This tool allows key information from a coversation with specific users to be stored in the system for future reference, while all other content in the conversation is lost once the session ends. This mechanism enables GPT to have some long-term memory, thereby improving the continuity across sessions and making conversations more personalized. As a result of this added bio tool feature, GPT's behavior becomes more human-like.

Finally we would like to extrapolate from the previous discussion on emergence. As we see an LLM with relatively simple structures and learning rules may exhibit unexpected complex behaviors, some of which mimic human-like understanding or even consciousness, when both the model size (number of parameters) and the training data size increase significantly. The phenomenon of emergence is not well understood, much like human consciousness, which is not well understood either. Could we, then, speculate that consciousness may also be some kind of emergent property of a super-complex system, the human brain, arising from relatively basic neural functions in an immensely complex hierarchical network structure containing 100 billion neurons connected by 100 trillion synapses?

The author heard a story at a conference on neural networks in the mid-'80s that went like this: When an elementary school teacher asked her class to name the most important inventions of the 20th century, her students mentioned the airplane, television, and computer. Then a boy shouted, "It is the thermos!" Puzzled, the teacher asked, "All the thermos does is keep water cold or hot, why is that so important?" The boy replied: "But how does it know when to keep the water cold and when to keep it hot?" (When asked to proofread this paragraph, ChatGPT made an additional comment: "This is a charming anecdote!" – an obvious emergent behavior!)

Now, many decades later, AI researchers are still grappling with the same question: *How does it know?*

21.11 Future Directions

Although modern LLMs have become powerful tools widely used in various areas such as education, scientific research, and medicine, they also face some significant challenges. One of the key issues is misinformation and hallucination, which is particularly problematic for applications requiring high accuracy, such as journalism, education, and legal documentation. Developing mechanisms to ensure the factual correctness of LLM outputs remains a major hurdle.

To address issues such as misinformation and hallucination, LLMs need to be improved to enhance their factual reliability. This will likely involve integrating external knowledge sources, such as databases and real-time web access, into their architecture. These enhancements could improve their ability to provide accurate information and reduce hallucinations. Fact-checking mechanisms and real-time data integration could also help ensure that outputs are more trustworthy.

Emergence in LLMs is an area of active research because it hints at deep, underlying principles about learning and intelligence that are still not fully understood. Additionally, the emergence of behaviors can be difficult to predict; researchers often do not know what capabilities will arise as models scale, leading to a kind of "black box" nature in how and why specific abilities develop. Key questions include:

- What specific factors contribute to the emergence of complex behaviors in LLMs?
- How does the scale of a model influence the types of capabilities that can emerge?
- What role does the architecture of the model play in facilitating or hindering emergence?
- How can we design training processes that guide the emergence of desired capabilities while minimizing undesirable ones?

The future of LLMs extends far beyond their current capabilities, encompassing both technical advancements and profound philosophical questions. As these models become more multimodal, personalized, and integrated with various technologies, they will continue to transform fields like education, healthcare, and scientific research. As these models evolve, they will also open new possibilities across a wide range of areas, such as automation of cognitive tasks and augmented creativity and innovation. At the same time, their development prompts deeper reflections on issues like understanding, consciousness, and their potential impact on society and human knowledge. The following are key areas where LLMs may drive significant change in the future, in terms of both the techinical development and some philosophical considerations.

- Multimodal Capabilities and Enhanced Contextual Understanding:

  LLMs are expected to evolve into multimodal systems, capable of processing and generating text, images, audio, and vide. This would enable richer, more interactive communication and problem-solving by interpreting and producing complex multimedia content.

  The ability to handle multiple modalities blurs the line between human sensory processing and machine interpretation. This raises questions about whether machines can achieve a deeper "understanding" of the world or if they are merely processing information in increasingly complex ways. What does it mean to "see" or "hear" when a machine can do these things?
- Personalization and Real-Time Learning

  As LLMs integrate user-specific data (while addressing privacy concerns), they will offer more personalized experiences. Real-time learning will allow them to adapt to new information dynamically, improving relevance and continuity in user interactions.

  Personalized learning systems challenge our notion of individuality and agency. As LLMs become more attuned to specific users, are they shaping users' preferences, or are users shaping the machine? How do we ensure the balance between personalization and privacy?
- Collaboration with Human Experts

  LLMs could become valuable collaborators for professionals, analyzing large datasets and generating insights that assist experts in fields like medicine, law, and engineering.

The role of machines as collaborators leads to reflections on human identity and work. As machines take on more sophisticated tasks, how do we define human expertise and creativity? Are we enhancing our capabilities, or are we becoming overly reliant on AI for innovation?

- Fact-Checking and Factual Reliability

  LLMs face the challenge of providing accurate information, especially in fields where misinformation or hallucination is problematic. Integrating external knowledge sources and real-time data could enhance their factual reliability and reduce errors.

  As machines become trusted sources of information, this raises questions about the nature of truth and authority. Should machines be held to the same standards of truth as humans, or is their understanding of "facts" inherently different? Can AI ever fully eliminate biases and inaccuracies, or will human-like flaws always persist?

- Integration with IoT and Accessibility Enhancements

  LLMs could be integrated with Internet of Things (IoT) devices, enhancing communication with smart technologies and enabling seamless interaction between users and their environment.

  The growing integration of AI with our physical surroundings challenges ideas about control and agency. As smart environments become more autonomous, where does human agency end and machine-driven decision-making begin? Do we risk ceding too much control to the systems that guide our lives?

- Emergence and the Black Box of AI Capabilities

  Emergent behaviors in LLMs often appear unexpectedly as models scale in size and complexity. Researchers do not always understand how or why specific capabilities develop, leading to a "black box" phenomenon.

  The unpredictability of emergent behaviors in AI mirrors broader questions about intelligence, consciousness, and free will. Can AI ever fully achieve human-like reasoning and decision-making, or is it confined to a mechanistic, black-box nature? Does emergence hint at a deeper intelligence within machines that we have yet to comprehend?

- Human Consciousness vs. Machine "Understanding"

  LLMs can generate responses that appear to exhibit understanding of nuanced and complex relationships, but they don't truly "understand" in the way humans do.

  This raises fundamental questions about the nature of consciousness. If machines can replicate behaviors associated with understanding or even emotions, how do we distinguish between human consciousness and machine intelligence? Could consciousness itself be an emergent property of sufficiently complex systems, both biological and artificial?

- Scientific Research and Creativity

LLMs may accelerate scientific discovery and creative processes by analyzing vast amounts of data, generating hypotheses, and aiding in the design of new concepts or artworks.

AI's role in creativity provokes deep questions about the essence of artistic expression. Can a machine truly be creative, or is creativity inherently tied to human emotion, experience, and intention? What role will humans play as machines become more involved in the arts and sciences?

In conclusion, by merging technical challenges with philosophical considerations, we get a holistic view of the future of LLMs, where technical advancements are not just questions of capability but also of meaning, ethics, and human identity. The interplay between these two dimensions will shape how LLMs evolve and how society responds to their integration into our lives.

21.12 Applications of Transformers Beyond NLP

Although the Transformer architecture was initially developed to tackle natural language processing (NLP) problems, it has emerged as a powerful tool across diverse fields much beyond NLP, due to its remarkable ability to effectively model complex relationships in sequential and spatial data. Here are some notable applications of transformers outside of NLP:

- Computer Vision

  Transformers have been widely used in computer vision, most notably with the *Vision Transformer (ViT)*, first introduced in *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, by Alexey Dosovitskiy, et al. (arXiv:2010.11929). ViT has achieved state-of-the-art performance in tasks such as image recognition and object detection. It treats image patches (e.g., 16×16 pixels) as token vectors, similar to words converted into imbedding vectors used in NLP. By processing these sequences of patches and using self-attention, the transformer learns to model complex spatial relationships between patches and capture both global context and local details.

  The output of the ViT depends on the specific task. For image classification (assuming there is only one object in the image), the output is typically a probability distribution over predefined classes (e.g., "cat," "dog," "car"). In image object detection, the transformer generates multiple output tokens, each corresponding to one of the objects within the image.

  The ViT is trained on labled data, where each object in the image is labeled by (1) its class identify, and (2) a bounding box (in terms of its center, together with its width and height). Based on the total loss, the difference between the ground truth labeling and the model output (combining both

classification and bounding box losses), backpropagation is carried out to adjust the model parameters, gradually reducing the total loss.

Unlike convolutional neural networks (CNNs), which inherently capture spatial relationships due to the local receptive fields, transformers lack this inductive bias. To compensate, ViT adds positional encodings to each patch vector. These encodings provide the model with information about the relative position of each patch in the image, helping it understand spatial relationships.

- Protein Structure Prediction

  Transformers have been used to predict protein structures, such as in AlphaFold by DeepMind (*Highly accurate protein structure prediction with AlphaFold*, by Jumper et al., Nature, 596(7873), 583–589). In this application, the amino acid sequences of proteins are treated similar to sentences in NLP. This transformer-based model, once trained on massive protein databases, can predict a protein's 3D structure based on its amino acid sequence. This is achieved by capturing both local (short-range) and global (long-range) dependencies within the sequence, which are essential for predicting folding patterns.

  AlphaFold is trained on data consisting of a large number of proteins with known amino acid sequences, each labeled by its experimentally determined 3D structure. The primary datasets include the Protein Data Bank (PDB), which provides high-quality structures derived from X-ray crystallography and cryo-electron microscopy, capturing a wide range of protein conformations. AlphaFold also leverages multiple sequence alignments (MSAs) and genomic data to extract evolutionary information, which is crucial for understanding conserved structural features across related sequences.

  Once trained, AlphaFold outputs several critical predictions for constructing a 3D structure for the sequence of a given protein. These include:
  - The distances between different parts of the protein and their relative orientations in space, which help to understand the protein's overall shape.
  - The angles of the protein's backbone structure, which define its overall folding pattern.
  - A complete 3D structure and shape in terms of the positions of all parts of the protein. This is a comprehensive representation that includes the spatial coordinates of the atoms in the protein, along with a visualization of the 3D structure.
  - Confidence scores for different parts of the protein and estimates regarding the accuracy of the model predictions.

  Transformers can also be fine-tuned to generate new protein sequences with desired structural or functional properties. By training on existing proteins, they can design sequences that are likely to adopt specific folds, leading to novel proteins with unique functions for therapeutic or industrial applications.

- Mathematics and Symbolic Reasoning

  Transformers have been adapted for symbolic math problems, including tasks such as integration, equation solving, and formal logic, where they have demonstrated significant promise in performing symbolic transformations and simplifications. These models utilize their attention mechanisms to capture intricate relationships between mathematical entities, enabling them to manipulate and reason about symbols effectively. For example, in integration tasks, transformers can learn to recognize patterns in integrands and apply known integration techniques, while in equation solving, they can systematically derive solutions by breaking down complex expressions into manageable components. Additionally, transformers have been applied to formal logic, where they can handle logical expressions and infer conclusions through symbolic reasoning. These applications are particularly valuable for automated theorem proving, as they allow for the verification of mathematical statements through logical deductions, and for symbolic computation, where they can facilitate the manipulation of mathematical expressions with high accuracy and efficiency.

- Speech and Audio Processing

  Transformers have demonstrated strong results in various tasks, including automatic speech recognition (ASR), speech synthesis, and audio classification. These models excel in handling audio data by treating it as a sequence of features, enabling them to capture temporal dependencies and intricate patterns within the audio signal. For instance, models like Wav2Vec 2.0 leverage transformers to process raw audio waveforms directly, employing a self-supervised learning approach that allows them to learn meaningful representations from unlabeled audio data. This capability enhances their performance in transcription tasks, achieving high accuracy by effectively distinguishing phonetic nuances and contextual information. In addition to ASR, transformers are also employed in speech synthesis, where they generate high-quality, natural-sounding speech by modeling the relationship between textual input and acoustic features. Furthermore, in audio classification tasks, transformers can analyze diverse audio signals, ranging from music to environmental sounds, by learning to identify distinctive features and patterns, leading to accurate classification outcomes.

In summary, the versatility of the Transformer architecture lies in its capacity to model dependencies across various types of sequential data, regardless of the domain. Its self-attention mechanism allows for the effective handling of high-dimensional inputs, capturing complex relationships, and enabling contextual understanding. This adaptability makes transformers suitable for solving intricate problems in widely different fields.

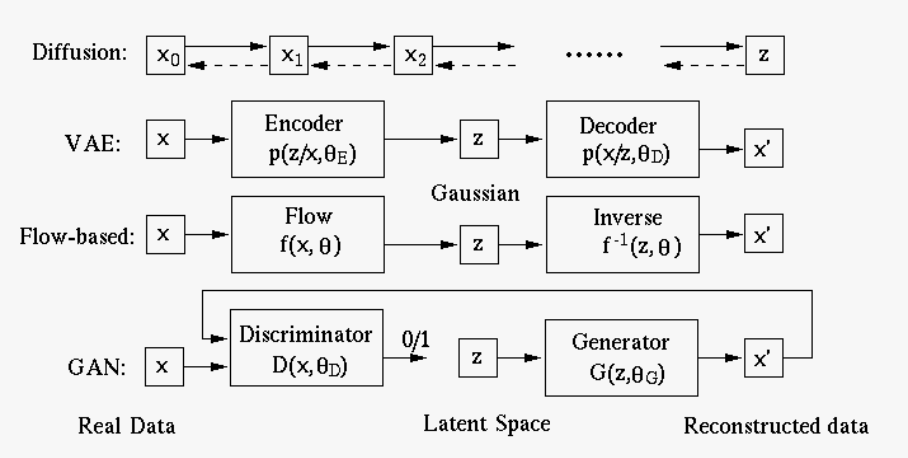


Figure 21.3 Four Generative Models

21.13 Generative Methods Beyond Transformers

While the Transformer architecture has been a groundbreaking development in large language modeling, numerous other generative learning methods have also emerged, such as those shown below. These generative models learn to approximate, explicitly or implicitly, the probability distribution over the given data (e.g., text, images, audio, and video), and generate new data samples that resemble to given data. This enables applications such as image and video synthesis, 3D scene reconstruction, and natural language generation.

All these models use neural network methods, leveraging deep learning networks trained through backpropagation based on unlabeled data, as their objective is to learn the structure and distribution of the input data to create realistic samples mimicing the input. Additionally, these models share a common general strategy: they learn to represent the given data in the real world (denoted by $\mathbf{x}$) in an internal hidden or latent space (denoted by $\mathbf{z}$), where the data is encoded in a more abstract or compressed form, making it more managable. The four types of models are illustrated as in Fig. 21.3, and discussed in more detail below.

- Diffusion Models

  Diffusion models operate through a two-step process: a forward diffusion process and a reverse diffusion process. In the forward process, noise is gradually added to the data across a series of time steps, progressively corrupting it until it becomes indistinguishable from pure noise. During training, the model learns to reverse this process by estimating the necessary adjustments to remove the noise incrementally. This is typically accomplished through a neural network that predicts the original data from the noisy input. Once trained, the model can generate new data by starting

with a random noise sample and iteratively applying the learned denoising steps, resulting in coherent outputs.

Diffusion models are generally trained on unlabeled data, because the model learns to generate data by progressively denoising a noisy sample until it resembles the original input data, which serves as the target for reconstruction. In each training iteration, the model's predicted denoising steps are compared with the actual data, and backpropagation is used to adjust the model's parameters, improving its ability to remove noise accurately. This iterative approach enables the model to learn a series of denoising steps that can eventually generate new, coherent data from pure noise once training is complete.

The diffusion models have been applied to generation of high-quality images and video based on textual descriptions, or even noise. They can also enhance the resolution of images, super-resolution, by refining low-resolution inputs.

- Variational Autoencoders

  Variational Autoencoders (VAEs) consist of two main components: an encoder and a decoder (similar to the autoencoder discussed in Section 18.3), both implemented as neural networks. The encoder learns to map the probability distribution of the input data to a distribution in a latent space, typically Gaussian parameterized by its mean and covariance. The decoder then draws samples from this latent space and maps them back to the data space, producing realistic outputs that resemble the original inputs. To capture the data's structures and to generate high-quality reconstructions, both the encoder and decoder are typically deep neural networks, especially for complex data like images, where architectures such as convolutional neural networks (Section 18.4) are commonly used.

  As the goal is to produce outputs that resemble the inputs (i.e., the input is the target for the output), VAEs are trained on unlabeled data. Backpropagation is used to minimize a loss function that combines two terms: the mean squared error between the reconstructed output and the original input, and the KL-divergence between the distribution in the latent space and the desired distribution, typically a Gaussian. Minimizing this loss function enables VAEs to balance data reconstruction accuracy and latent space regularization, ensuring that samples drawn from the latent space are realistic and diverse.

  Similar to autoencoders, VAEs can be used for dimensionality reduction. However, unlike traditional autoencoders, VAEs are primarily used to generate new, realistic output that exhibit characteristics similar to the input, instead the exact replicas of the input data. This capability makes VAEs particularly useful in applications where generating diverse, yet coherent, samples is essential. For example, in image generation, a VAE trained on a dataset of human faces can produce new, realistic-looking faces with dif-

ferent features, expressions, or poses, while retaining a resemblance to the types of faces it was trained on.

- Flow-based Models

  Flow-based models learn to transform a known and mathematically tractable distribution, typically Gaussian, in a latent space (also called Gaussian space), into a more complex distribution in data space that resembles the actual data distributions. This is achieved progressively by a sequence of invertible transformations, called *flows*, implemented by a multi-layer neural network, with each layer corresponding to one of the transformations. Since each transformation is invertible, the model can be reversed to map data points from the data space back to the Gaussian latent space.

  During training, samples from the training set (data space) are passed through the model to be mapped to the Gaussian in the latent space for the calculation of the likelihood of the data points under the Gaussian distribution (in terms of ite mean and covariance). This represents how well the Gaussian models the real data. Then, backpropagation is used to minimize a loss function, the negative log-likelihood, so that the model parameters, the weights in the network, are modified to enable the Gaussian to better model the data, thereby improving its performance in generating realistic samples.

  Once trained, a flow-based model can generate new data by passing random samples from the Gaussian in the latent space through the inverse of the transformation network (the flow) to obtain new data points in the data space. These generated data should exhibit the same characteristics as the original data, though they are not replicas.

  Flow-based models can generate high-fidelity images useful for applications like image generation, denoising and super-resolution. They can also be used for text and video generation.

- Generative Adversarial Networks

  Generative adversarial networks (GANs) consist of two deep neural networks: a generator and a discriminator, which work in opposition to each other in a process known as adversarial training. The generator transforms random noise into synthetic (fake) data samples that resemble real data, while the discriminator takes both real and fake data samples as input and generate a probability score as output to indicate whether the input is real or fake.

  During training, the generator aims to maximize the probability of the discriminator making a mistake, while the discriminator strives to accurately classify the inputs. This competition between the two networks leads to progressively improved performance on both sides, with the generator producing increasingly realistic samples. Training is complete once it reaches the *Nash equilibrium*, a game theory concept, where neither the discriminator nor the generator can improve its performance any further. At this point, the discriminator can no longer distinguish between the fake

data samples produced by the generator and the real samples from the dataset, indicating that the model has successfully learned the underlying data distribution,

Specifically, the discriminator aims to maximize the expectation of its output, $D(\mathbf{x}, \theta_D)$ when the input $\mathbf{x}$ is a real sample from dataset, and to minimize the expectation of $D(G(\mathbf{z}), \theta_G)$ (or equivalently, maximizing $1 - D(G\mathbf{z}, \theta_G)$) when the input is a fake sample from the generator. Thus, the discriminator seeks to maximize the following objective function:

$$J(D, G) = E_{p(x)}[\log D(\mathbf{x}, \theta_D)] + E_{p(z)}[\log(1 - D(G(\mathbf{z}, \theta_G)))] \quad (21.4)$$

On the other hand, the generator aims to achieve its goal of "fooling" the discriminator into classifying its generated (fake) samples as real (from the dataset). To achieve this, it seeks to minimize $J(D, G)$, the objective that the discriminator is trying to maximize. If the generator successfully minimizes $J(D, G)$, it will cause $D(G(z))$ to approach 1 for generated samples, effectively "tricking" the discriminator. This leads to a situation where the generator's fake samples are indistinguishable from real samples, thereby achieving the goal of the GAN. Consequently, the overall objective of the GAN is expressed as $\min_G \max_D J(D, G)$. Backpropagation is then used to update the parameters θ_G and θ_D, the weights of both the generator and discriminator networks, based on the gradients of $J(D, G)$.

Once trained, GANs can generate new data by sampling random noise from a predefined distribution, typically a Gaussian or uniform distribution, and passing this noise through the generator network to transform it into new synthetic data points that resemble the original training data. GANs can generate high-quality, realistic images, audio, and video, that align with the learned data distribution.

In summary, these recent generative machine learning methods showcase the evolving landscape of generative AI. Each method offers unique mechanisms for generating high-quality data across diverse applications, from image and video synthesis to natural language processing and 3D reconstruction. Their advancements signify a substantial leap in the capabilities of generative models, enhancing their applicability in real-world scenarios.

Appendix A A Review of Linear Algebra

A.1 Inner Product Space

A.1.1 Vector Space

- **Vector space**

A *vector space* over field F is a set V which is closed under the following two operations of addition and scalar multiplication defined for its members (called vectors), i.e., the results of the operations are also members of V.

1.The vector addition that maps any two vectors $\mathbf{x}, \mathbf{y} \in V$ to another vector $\mathbf{x}+\mathbf{y} \in V$ satisfying the following properties:

- Commutativity: $\mathbf{x}+\mathbf{y} = \mathbf{y}+\mathbf{x}$.
- Associativity: $\mathbf{x}+(\mathbf{y}+\mathbf{z}) = (\mathbf{x}+\mathbf{y})+\mathbf{z}$.
- Existence of zero: there is a vector $\mathbf{0} \in V$ such that: $\mathbf{0}+\mathbf{x} = \mathbf{x}+\mathbf{0} = \mathbf{x}$.
- Existence of inverse: for any vector $\mathbf{x} \in V$, there is another vector $-\mathbf{x} \in V$ such that $\mathbf{x}+(-\mathbf{x}) = \mathbf{0}$.

2.The scalar multiplication that maps a vector $\mathbf{x} \in V$ and scalar $a \in F$ (F can be a real or complex field) to another vector $a\mathbf{x} \in V$ with the following properties:

- $a(\mathbf{x}+\mathbf{y}) = a\mathbf{x}+a\mathbf{y}$.
- $(a+b)\mathbf{x} = a\mathbf{x}+b\mathbf{x}$.
- $ab\mathbf{x} = a(b\mathbf{x})$.
- $1\mathbf{x} = \mathbf{x}$.

where $\mathbf{0} = [0, \cdots, 0]^T$ and $\mathbf{1} = [1, \cdots, 1]^T$ are two constant vectors.

A subset W of V is a subspace of V, denoted by $W \subseteq V$, if it is also a vector space, i.e., it is closed under the same operations defined in V:

1.The zero vector $\mathbf{0}$ of V must be in W (the zero vector is unique in $\mathbf{V}$, which $\mathbf{W}$ must have).

2.For any $\mathbf{x}$, $\mathbf{y} \in W$, $\mathbf{x}+\mathbf{y} \in W$.

3.For any $\mathbf{x} \in W$, $a\mathbf{x} \in W$.

Listed below is a set of typical vector spaces:

- n-D vector space $\mathbb{R}^n$ or $\mathbb{C}^n$

This space contains all n-D vectors expressed as an n-tuple, an ordered list of n elements (or components):

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = [x_1, x_2, \ldots, x_n]^T, \tag{A.1}$$

which can be used to represent a discrete signal containing n samples. We will always represent a vector as a vertical or column vector, or the transpose of a horizontal or row vector. The space is denoted by either $\mathbb{C}^n$ if the elements are complex $x_i \in \mathbb{C}$, or $\mathbb{R}^n$ if they are all real $x_i \in \mathbb{R}$ $(i = 1, \ldots, n)$.

A subspace of $\mathbb{R}^n$ can be a $\mathbb{R}^m$ $(m < n)$ that passes origin (zero). For example, any 2-D plane passing through the origin of a 3-D space is its subspace. However, if the 2-D plane does not pass through the origin, it is not a subspace. Also, as 3-D cube or sphere centered at the origin is not a subspace, as it is not closed under the operations of addition and scalar multiplication.

– A vector space can be defined to contain all $m \times n$ matrices composed of n m-D column vectors:

$$\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_n] = \begin{bmatrix} a_{11} & a_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}, \tag{A.2}$$

where the ith column is an m-D vector $\mathbf{a}_i = [x_{1i}, \ldots, x_{mi}]^T$. Such a matrix can be converted to an mn-D vector by cascading all of the column (or row) vectors.

– l^2 space:

The concept of an n-D space $\mathbb{R}^n$ or $\mathbb{C}^n$ can be generalized by allowing the dimension n to become to infinity so that a vector in the space becomes a sequence $\mathbf{x} = [\ldots, x_i, \ldots]^T$ for $0 \leq i < \infty$ or $-\infty < i < \infty$. If all vectors are square-summable, the space is denoted by l^2. All discrete energy signals are vectors in l^2.

– $\mathcal{L}^2$ space:

A vector space can also be a set of real or complex valued continuous functions $x(t)$ defined over either a finite range such as $0 \leq t < T$, or an infinite range $-\infty < t < \infty$. If all functions are square-integrable, the space is denoted by $\mathcal{L}^2$. All continuous energy signals are vectors in $\mathcal{L}^2$.

Note that the term "vector", generally denoted by $\mathbf{x}$ in the following, may be interpreted in two different ways. First, in the most general sense, it represents a member of a vector space, such as any of the vector spaces

considered above; e.g., a function $\mathbf{x} = x(t) \in \mathcal{L}^2$. Second, in a more narrow sense, it can also represent a tuple of n elements, an n-D vector $\mathbf{x} = [x_1, \ldots, x_n]^T \in \mathbb{C}^n$, where n may be infinity. It should be clear what a vector $\mathbf{x}$ represents from the context.

- **Linear independence**

  A set of vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$ are linearly independent if none of them can be represented as a linear combination of the others.

  Theorem: For any set of linearly independent vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$, if their linear combination is zero

$$\sum_{j=1}^{n} c_j \mathbf{v}_j = \mathbf{0} \tag{A.3}$$

  then all coefficients must be zero $c_1 = \cdots = c_n = 0$, or $\mathbf{c} = \mathbf{0}$.

  Proof: Assume $c_k \neq 0$, then we would get

$$\mathbf{v}_k = -\frac{1}{c_k} \sum_{j=1,\ i \neq k}^{n} c_j \mathbf{v}_j \tag{A.4}$$

  i.e., $\mathbf{v}_k$ is a linear combination of the remaining $n-1$ vectors, in contradiction with the assumption that they are independent.

  In particular, in the n-D Euclidean space, the n vectors can be written as $\mathbf{v}_j = [v_{1j}, \cdots, v_{nj}]^T$ $(j = 1, \cdots, n)$, and their linear combination

$$\sum_{j=1}^{n} c_j \mathbf{v}_j = [\mathbf{v}_1, \cdots, \mathbf{v}_n] \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} = \mathbf{V}\mathbf{c} \tag{A.5}$$

  For this to be a zero vector, i.e., for the homogeneous equation $\mathbf{V}\mathbf{c} = \mathbf{0}$ to hold, the coefficient vector $\mathbf{c}$ has to be zero, as $\mathbf{V}$ is a full rank matrix. QED

- **Convex combination**

  The *convex combination* of n vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n$ is their sum weighted by coefficients $\{c_1, \cdots, c_n\}$ that add up to 1:

$$\sum_{i=1}^{n} c_i \mathbf{x}_i, \qquad \sum_{i=1}^{n} c_i = 1 \tag{A.6}$$

  The *convex hull* of these points is the set of all their convex combinations. For example, the convex hull of three points in a plane is the triangle formed by these points as vertices, in which any point is a convex combination of the three vertices.

- **Inner product**

  An *inner product* in a vector space V is a function that maps two vectors $\mathbf{x}, \mathbf{y} \in V$ to a scalar $\langle \mathbf{x}, \mathbf{y} \rangle \in \mathbb{C}$ or $\mathbb{R}$ and satisfies the following conditions:

– Positive definiteness:

$$\langle \mathbf{x}, \mathbf{x} \rangle \ \geq \ 0, \qquad \langle \mathbf{x}, \mathbf{x} \rangle = 0 \ \ \text{iff} \ \ \mathbf{x} = \mathbf{0}. \tag{A.7}$$

– Conjugate symmetry:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \overline{\langle \mathbf{y}, \mathbf{x} \rangle}. \tag{A.8}$$

If the vector space is real, the inner product becomes *symmetric*:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{y}, \mathbf{x} \rangle. \tag{A.9}$$

– Linearity in the first variable:

$$\langle a\mathbf{x} + b\mathbf{y}, \mathbf{z} \rangle = a\langle \mathbf{x}, \mathbf{z} \rangle + b\langle \mathbf{y}, \mathbf{z} \rangle, \tag{A.10}$$

where $a, b \in \mathbb{C}$. The linearity does not apply to the second variable unless the coefficients are real $a, b \in \mathbb{R}$:

$$\langle \mathbf{x}, a\mathbf{y} + b\mathbf{z} \rangle = \overline{\langle a\mathbf{y} + b\mathbf{z}, \mathbf{x} \rangle} = \overline{a\langle \mathbf{y}, \mathbf{x} \rangle + b\langle \mathbf{z}, \mathbf{x} \rangle} = \overline{a}\langle \mathbf{x}, \mathbf{y} \rangle + \overline{b}\langle \mathbf{x}, \mathbf{z} \rangle$$
$$\neq a\langle \mathbf{x}, \mathbf{y} \rangle + b\langle \mathbf{x}, \mathbf{z} \rangle$$

In the special case when $b = 0$, we have

$$\langle a\mathbf{x}, \mathbf{y} \rangle = a\langle \mathbf{x}, \mathbf{y} \rangle, \qquad \langle \mathbf{x}, a\mathbf{y} \rangle = \overline{a}\langle \mathbf{x}, \mathbf{y} \rangle. \tag{A.11}$$

More generally we have

$$\left\langle \sum_n c_n \mathbf{x}_n, \mathbf{y} \right\rangle = \sum_n c_n \langle \mathbf{x}_n, \mathbf{y} \rangle, \qquad \left\langle \mathbf{x}, \sum_n c_n \mathbf{y}_n \right\rangle = \sum_n \overline{c}_n \langle \mathbf{x}, \mathbf{y}_n \rangle. \tag{A.12}$$

An *inner product space* is a vector space with inner product defined. In particular, when the inner product is defined, $\mathbb{C}^n$ is called a *unitary space* and $\mathbb{R}^n$ is called a *Euclidean space*.

Some examples of the inner product are listed below:

– In an n-D vector space, the inner product, also called the *dot product*, of two vectors $\mathbf{x} = [x_1, \ldots, x_n]^T$ and $\mathbf{y} = [y_1, \ldots, y_n]^T$ is defined as

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \overline{\mathbf{y}} = \mathbf{y}^* \mathbf{x} = [x_1, \ldots, x_n] \begin{bmatrix} \overline{y}_1 \\ \vdots \\ \overline{y}_n \end{bmatrix} = \sum_{i=1}^{n} x_i \overline{y}_i, \tag{A.13}$$

where $\mathbf{y}^* = \overline{\mathbf{y}}^T$ is the conjugate transpose of $\mathbf{y}$. If $\mathbf{y} = \overline{\mathbf{y}}$ is real, then $\mathbf{y}^* = \mathbf{y}^T$.

– In a space of 2-D matrices containing $m \times n$ elements, the inner product of two matrices $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$\langle \mathbf{A}, \mathbf{B} \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} \overline{b}_{ij}. \tag{A.14}$$

When the column (or row) vectors of $\mathbf{A}$ and $\mathbf{B}$ are concatenated to

form two mn-D vectors, their inner product takes the same form as that of two n-D vectors.

– In a function space, the inner product of two function vectors $\mathbf{x} = x(t)$ and $\mathbf{y} = y(t)$ is defined as

$$\langle x(t), y(t)\rangle = \int_a^b x(t)\overline{y(t)}\, dt = \overline{\int_a^b y(t)\overline{x(t)}\, dt} = \overline{\langle y(t), x(t)\rangle}. \quad \text{(A.15)}$$

– The covariance of two random variables x and y can be considered as an inner product

$$\langle x,\, y\rangle = \sigma_{xy}^2 = E[(x-\mu_x)\overline{(y-\mu_y)}] = E(x\overline{y}) - \mu_x\overline{\mu}_y \quad \text{(A.16)}$$

Specially when $\mu_x = \mu_x = 0$, we have

$$\langle x, y\rangle = E[x\overline{y}]. \quad \text{(A.17)}$$

- **Vector norm**

In general, the norm $||\mathbf{x}||$ of a vector $\mathbf{x} \in V$ is a non-negative real scalar that measures its size or length. $||\mathbf{x}|| = 0$ if and only if $\mathbf{x} = \mathbf{0}$. There exist different definitions for vector norm, as shown in Section A.4.1. A vector $\mathbf{x}$ is *normalized* (becomes a *unit vector*) if $||\mathbf{x}|| = 1$. Any given vector can be normalized when divided by its own norm: $\mathbf{x}/||\mathbf{x}||$.

The most widely used *2-norm* of a vector is defined as

$$||\mathbf{x}|| = \sqrt{\langle \mathbf{x}, \mathbf{x}\rangle} = \langle \mathbf{x}, \mathbf{x}\rangle^{1/2}, \quad \text{(A.18)}$$

The vector norm squared $||\mathbf{x}||^2 = \langle \mathbf{x}, \mathbf{x}\rangle$ can be considered as the energy of the vector. In particular, in an n-D unitary space, the 2-norm of a vector $\mathbf{x} = [x_1, \ldots, x_n]^T \in \mathbb{C}^n$ is:

$$||\mathbf{x}|| = \sqrt{\langle \mathbf{x}, \mathbf{x}\rangle} = \sqrt{\mathbf{x}^T\overline{\mathbf{x}}} = \sqrt{\mathbf{x}^*\mathbf{x}} = \left(\sum_{i=1}^n x_i\overline{x}_i\right)^{1/2} = \left(\sum_{i=1}^n |x_i|^2\right)^{1/2}. \quad \text{(A.19)}$$

The total energy contained in this vector is its norm squared:

$$\mathcal{E} = ||\mathbf{x}||^2 = \langle \mathbf{x}, \mathbf{x}\rangle = \sum_{i=1}^n |x_i|^2. \quad \text{(A.20)}$$

Similarly, in a function space, the norm of a function vector $\mathbf{x} = x(t)$ is defined as

$$||\mathbf{x}|| = \left(\int_a^b x(t)\overline{x(t)}\ dt\right)^{1/2} = \left(\int_a^b |x(t)|^2\ dt\right)^{1/2}, \quad \text{(A.21)}$$

where the lower and upper integral limits $a < b$ are two real numbers, which may be extended to all real values $\mathbb{R}$ in the entire real axis $-\infty < t < \infty$.

This norm exists only if the integral converges to a finite value; i.e., $x(t)$ is an *energy signal* containing finite energy:

$$\int_{-\infty}^{\infty} \left|x(t)\right|^2 dt < \infty. \tag{A.22}$$

All such functions $x(t)$ satisfying the above are square-integrable, and they form a function space denoted by $\mathcal{L}^2(\mathbb{R})$.

- **Cauchy-Schwarz inequality**

  The Cauchy-Schwarz inequality holds for any two vectors $\mathbf{x}$ and $\mathbf{y}$ in an inner product space V:

$$|\langle \mathbf{x}, \mathbf{y}\rangle|^2 \ \leq \ \langle \mathbf{x}, \mathbf{x}\rangle \langle \ \mathbf{y}, \mathbf{y}\rangle; \qquad \text{i.e.,} \quad 0 \leq |\langle \mathbf{x}, \mathbf{y}\rangle| \leq ||\mathbf{x}||\ ||\mathbf{y}|| \tag{A.23}$$

  Proof: If either $\mathbf{x}$ or $\mathbf{y}$ is zero, $\langle \mathbf{x}, \mathbf{y}\rangle = 0$, the theorem holds (an equality). Otherwise, we consider the following inner product:

$$\langle \mathbf{x} - \lambda\mathbf{y}, \mathbf{x} - \lambda\mathbf{y}\rangle = ||\mathbf{x}||^2 - \overline{\lambda}\langle \mathbf{x}, \mathbf{y}\rangle - \lambda\langle \mathbf{y}, \mathbf{x}\rangle + |\lambda|^2||\mathbf{y}||^2 \geq 0, \tag{A.24}$$

  where $\lambda \in \mathbb{C}$ is an arbitrary complex number, which can be assumed to be:

$$\lambda = \frac{\langle \mathbf{x}, \mathbf{y}\rangle}{||\mathbf{y}||^2}, \quad \text{then} \quad \overline{\lambda} = \frac{\langle \mathbf{y}, \mathbf{x}\rangle}{||\mathbf{y}||^2}, \quad |\lambda|^2 = \frac{|\langle \mathbf{x}, \mathbf{y}\rangle|^2}{||\mathbf{y}||^4}. \tag{A.25}$$

  Substituting these into the previous equation we get

$$||\mathbf{x}||^2 - \frac{|\langle \mathbf{x}, \mathbf{y}\rangle|^2}{||\mathbf{y}||^2} \geq 0; \qquad \text{i.e.,} \quad |\langle \mathbf{x}, \mathbf{y}\rangle| \leq ||\mathbf{x}||\ ||\mathbf{y}||. \tag{A.26}$$

  The equation holds only if $\mathbf{x} - \lambda\mathbf{y} = 0$ or $\mathbf{x} = \lambda\mathbf{y}$, i.e., the two vectors are linearly dependent.

- **Distance between two vectors**

  The distance $d(\mathbf{x}, \mathbf{y})$ between two vectors $\mathbf{x}$ and $\mathbf{y}$ is a real constant that satisfies:
  - $d(\mathbf{x}, \mathbf{y}) = 0$ iff $\mathbf{x} = \mathbf{y}$
  - $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$
  - $d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{z}, \mathbf{y})$

  In an inner-product space in which the inner product $\langle \mathbf{x}, \mathbf{y}\rangle$ between any two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined, the distance between the two points can be defined as the norm of the difference between the two points:

$$d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}|| = \sqrt{\langle (\mathbf{x} - \mathbf{y}), (\mathbf{x} - \mathbf{y})\rangle} \tag{A.27}$$

  The norm of a vector can now be seen as its distance to the origin $\mathbf{0}$ of the space: $||\mathbf{x}|| = ||\mathbf{x} - \mathbf{0}||$.

  A vector space V is a *metric space* if a distance (or *metric*) $d(\mathbf{x}, \mathbf{y})$ between any two vector (two points) $\mathbf{x}$ and $\mathbf{y}$ is defined.

- **Angle between two vectors**

  The *angle* between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$\theta = \cos^{-1}\left(\frac{\langle \mathbf{x}, \mathbf{y}\rangle}{||\mathbf{x}||\ ||\mathbf{y}||}\right). \tag{A.28}$$

Now the inner product of $\mathbf{x}$ and $\mathbf{y}$ can also be written as

$$\langle \mathbf{x}, \mathbf{y} \rangle = ||\mathbf{x}||\ ||\mathbf{y}|| \cos\theta \leq ||\mathbf{x}||\ ||\mathbf{y}|| \tag{A.29}$$

This is the Cauchy-Schwarz inequality. In particular,

– If $\theta = 0$, $\cos\theta = 1$, then $\mathbf{x}$ and $\mathbf{y}$ are colinear or linearly dependent, and the inner product is maximized:

$$\langle \mathbf{x}, \mathbf{y} \rangle = ||\mathbf{x}||\ ||\mathbf{y}|| \tag{A.30}$$

i.e., the Cauchy-Schwarz inequality becomes an equality.

– If $0 < \theta < \pi/2$, $0 < \cos\theta < 1$, we get the Cauchy-Schwarz inequality:

$$\langle \mathbf{x}, \mathbf{y} \rangle < ||\mathbf{x}||\ ||\mathbf{y}|| \tag{A.31}$$

– If $\theta = \pi/2$, $\cos\theta = 0$, then $\mathbf{x}$ and $\mathbf{y}$ are orthogonal to each other, and the inner product is minimized:

$$\langle \mathbf{x}, \mathbf{y} \rangle = 0 \tag{A.32}$$

Two vectors $\mathbf{x}$ and $\mathbf{y}$ are *orthogonal* or *perpendicular* to each other, denoted by $\mathbf{x} \perp \mathbf{y}$, if their inner product is zero $\langle \mathbf{x}, \mathbf{y} \rangle = 0$, i.e., the angle between them is $\theta = \cos^{-1} 0 = \pi/2$.

- **Projection**

The *orthogonal projection* of a vector $\mathbf{x} \in V$ onto another vector $\mathbf{y} \in V$ is defined as a vector

$$\mathbf{p}_y(\mathbf{x}) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{||\mathbf{y}||} \frac{\mathbf{y}}{||\mathbf{y}||} = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\langle \mathbf{y}, \mathbf{y} \rangle} \mathbf{y} = ||\mathbf{x}|| \cos\theta \frac{\mathbf{y}}{||\mathbf{y}||}, \tag{A.33}$$

where $\mathbf{y}/||\mathbf{y}||$ is the unit vector along the direction of $\mathbf{y}$. In particular, if $\mathbf{y}$ is normalized with $||\mathbf{y}|| = 1$, then

$$\mathbf{p}_y(\mathbf{x}) = \langle \mathbf{x}, \mathbf{y} \rangle\ \mathbf{y} = ||\mathbf{x}|| \cos\theta\ \mathbf{y} \tag{A.34}$$

Note that

$$||\mathbf{p}_y(\mathbf{x})|| = ||\mathbf{x}|| \cos\theta \tag{A.35}$$

If only the magnitude of the projection is of interest, the unit vector $\mathbf{y}/||\mathbf{y}||$ can be dropped:

$$p_y(\mathbf{x}) = \langle \mathbf{x}, \mathbf{y} \rangle / ||\mathbf{y}|| \tag{A.36}$$

- **Cauchy space**

A sequence of points in a metric space $x_0, x_1, x_2, \cdots$ is a *Cauchy sequence* if it converges, i.e., for any $\epsilon > 0$, there exists an integer $N > 0$ so that the following is true for any $m, n > N$:

$$d(x_m, x_n) < \epsilon \tag{A.37}$$

If the limit of any Cauchy sequence of points in the space is also in the space, the space is *complete*, referred to as a *Cauchy space*.

A.1.2 Orthogonal Basis and Gram-Schmidt Process

Assume the n-D vector space $\mathbb{R}^n$ or $\mathbb{C}^n$ is spanned by a set of n independent basis vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$, not necessarily orthogonal, so that any vector $\mathbf{x} \in \mathbb{R}^n$ can be represented as a linear combination of the basis vectors:

$$\mathbf{x} = \sum_{i=1}^{n} c_i \mathbf{v}_i = \begin{bmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_n \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} = \mathbf{V}\mathbf{c} \tag{A.38}$$

where $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_n]$ is an $n \times n$ matrix composed of the n vectors and the coefficients in $\mathbf{c} = [c_1, \cdots, c_n]^T$ can be found by solving the linear equation system to get $\mathbf{c} = \mathbf{V}^{-1}\mathbf{x}$ with complexity is $O(n^3)$.

These linearly independent vectors $\mathbf{v}_1, \cdots, \mathbf{v}_n$ can be converted into a set of orthogonal vectors $\{\mathbf{u}_1, \cdots, \mathbf{u}_n\}$ satisfying $\mathbf{u}_i^T \mathbf{u}_j = 0$ $(i \neq j)$ by the following *Gram-Schmidt process*:

$$\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1 \\
\mathbf{u}_2 &= \mathbf{v}_2 - \mathbf{p}_{\mathbf{u}_1}(\mathbf{v}_2) = \mathbf{v}_2 - \left(\frac{\mathbf{v}_2^T \mathbf{u}_1}{\mathbf{u}_1^T \mathbf{u}_1}\right) \mathbf{u}_1 \\
\mathbf{u}_3 &= \mathbf{v}_3 - \mathbf{p}_{\mathbf{u}_1}(\mathbf{v}_3) - \mathbf{p}_{\mathbf{u}_2}(\mathbf{v}_3) = \mathbf{v}_3 - \left(\frac{\mathbf{v}_3^T \mathbf{u}_1}{\mathbf{u}_1^T \mathbf{u}_1}\right) \mathbf{u}_1 - \left(\frac{\mathbf{v}_3^T \mathbf{u}_2}{\mathbf{u}_2^T \mathbf{u}_2}\right) \mathbf{u}_2 \\
&\cdots\cdots\cdots \\
\mathbf{u}_k &= \mathbf{v}_k - \sum_{i=1}^{k-1} \mathbf{p}_{\mathbf{u}_i}(\mathbf{v}_k) = \mathbf{v}_k - \sum_{i=1}^{k-1} \left(\frac{\mathbf{v}_k^T \mathbf{u}_i}{\mathbf{u}_i^T \mathbf{u}_i}\right) \mathbf{u}_i
\end{aligned} \tag{A.39}$$

where $\mathbf{p}_{\mathbf{u}_i}(\mathbf{v}_k)$ is the projection of $\mathbf{v}_k$ onto $\mathbf{u}_i$:

$$\mathbf{p}_{\mathbf{u}_i}(\mathbf{v}_k) = \left(\frac{\mathbf{v}_k^T \mathbf{u}_i}{\mathbf{u}_i^T \mathbf{u}_i}\right) \mathbf{u}_i \tag{A.40}$$

We see that $\mathbf{u}_2$ so obtained is indeed orthogonal to all $\mathbf{u}_1$

$$\mathbf{u}_2^T \mathbf{u}_1 = \left[\mathbf{v}_2 - \left(\frac{\mathbf{v}_2^T \mathbf{u}_1}{\mathbf{u}_1^T \mathbf{u}_1}\right) \mathbf{u}_1\right]^T \mathbf{u}_1 = \mathbf{v}_2^T \mathbf{u}_1 - \frac{\mathbf{v}_2^T \mathbf{u}_1}{\mathbf{u}_1^T \mathbf{u}_1} \mathbf{u}_1^T \mathbf{u}_1 = 0 \tag{A.41}$$

and, by mathematical induction, we can further prove that $\mathbf{u}_k^T \mathbf{u}_i = 0$ for all $i = 1, \cdots, k-1$. In other words, $\mathbf{u}_k$ is the component of $\mathbf{v}_k$ that is orthogonal to all previously found orthogonal vectors $\mathbf{u}_1, \cdots, \mathbf{u}_{k-1}$.

Given a set of orthogonal basis $\{\mathbf{u}_1, \cdots, \mathbf{u}_n\}$ satisfying $\langle \mathbf{u}_i, \mathbf{u}_j \rangle = \mathbf{u}_i^T \mathbf{u}_j = 0$ for any $i \neq j$, we can again represent any given vector $\mathbf{x}$ as a linear combination of these basis as

$$\mathbf{x} = \sum_{i=1}^{n} d_i \mathbf{u}_i \tag{A.42}$$

The coefficients d_i can now be obtained easily. Premultiplying $\mathbf{u}_j^T$ on both sides

of the equation above, we get

$$\mathbf{u}_j^T\mathbf{x} = \mathbf{u}_j^T\left(\sum_{i=1}^{n} d_i\mathbf{u}_i\right) = \sum_{i=1}^{n} d_i\mathbf{u}_j^T\mathbf{u}_i = d_j\mathbf{u}_j^T\mathbf{u}_j \tag{A.43}$$

Solving for c_j we get

$$d_j = \frac{\mathbf{u}_j^T\mathbf{x}}{\mathbf{u}_j^T\mathbf{u}_j}, \qquad j = 1, \cdots, n \tag{A.44}$$

As the n coefficients are decoupled, each of them can be obtained separately with linear computational complexity $O(n)$ with total complexity $O(n^2)$ for all n of them. The complexity is reduced to $O(n^2)$ from $O(n^3)$ for solving the equation system $\mathbf{c} = \mathbf{V}^{-1}\mathbf{b}$, needed in the case where the basis is not orthogonal. This is the reason why orthogonal bases are preferred in general. Now the vector $\mathbf{x}$ can be written as

$$\mathbf{x} = \sum_{i=1}^{n} d_i\mathbf{u}_i = \sum_{i=1}^{n}\left(\frac{\mathbf{u}_j^T\mathbf{x}}{\mathbf{u}_j^T\mathbf{u}_j}\right)\mathbf{u}_i = \sum_{i=1}^{n}\mathbf{p}_{\mathbf{u}_i}(\mathbf{x}) \tag{A.45}$$

where the ith term of the summation above is simply the projection $\mathbf{p}_{\mathbf{u}_i}(\mathbf{x})$ of $\mathbf{x}$ onto the ith basis vector $\mathbf{u}_i$.

The concept of orthogonal vectors satisfying $\mathbf{u}^T\mathbf{v} = 0$ can be generalized to *conjugate vectors* that satisfy $\mathbf{u}^T\mathbf{A}\mathbf{v} = 0$ with respect to a symmetric matrix $\mathbf{A} = \mathbf{A}^T$. This can be expressed in the form of inner product

$$\mathbf{u}^T\mathbf{A}\mathbf{v} = (\mathbf{A}\mathbf{v})^T\mathbf{u} = \mathbf{v}^T\mathbf{A}\mathbf{u} \tag{A.46}$$

Conjugate vectors with respect to $\mathbf{A}$ can also be considered as orthogonal to each other with respect to $\mathbf{A}$. Two orthogonal vectors satisfying $\mathbf{u}^T\mathbf{v} = 0$ can be considered as a special case of conjugate vectors with respect to $\mathbf{A} = \mathbf{I}$.

Based on this generalized orthogonality, we can also define the projection of $\mathbf{v}$ onto $\mathbf{u}$ with respect to $\mathbf{A}$ as:

$$\mathbf{p}_{\mathbf{u}}(\mathbf{v}) = \left(\frac{\mathbf{u}^T\mathbf{A}\mathbf{v}}{\mathbf{u}^T\mathbf{A}\mathbf{u}}\right)\mathbf{u} \tag{A.47}$$

Given a set of A-conjugate basis vectors $\{\mathbf{d}_1, \cdots, \mathbf{d}_n\}$ satisfying $\mathbf{d}_i^T\mathbf{A}\mathbf{d}_j = 0$ for any $i \neq j$, we can represent any $\mathbf{x}$ as

$$\mathbf{x} = \sum_{i=1}^{n} a_i\mathbf{d}_i \tag{A.48}$$

The coefficients a_i can be obtained by pre-multiplying $\mathbf{d}_j^T\mathbf{A}$ on both sides:

$$\mathbf{d}_j^T\mathbf{A}\mathbf{x} = \sum_{i=1}^{n} a_i\mathbf{d}_j^T\mathbf{A}\mathbf{d}_i = a_j\mathbf{d}_j^T\mathbf{A}\mathbf{d}_j \tag{A.49}$$

Solving for a_j we get

$$a_j = \frac{\mathbf{d}_j^T\mathbf{A}\mathbf{x}}{\mathbf{d}_j^T\mathbf{A}\mathbf{d}_j}, \qquad j = 1, \cdots, n \tag{A.50}$$

and $\mathbf{x}$ is now represented as the sum of its A-projections onto the n basis vectors $\{\mathbf{d}_1, \cdots, \mathbf{d}_n\}$:

$$\mathbf{x} = \sum_{i=1}^{n} a_i \mathbf{d}_i = \sum_{i=1}^{n} \left(\frac{\mathbf{d}_i^T \mathbf{A}\mathbf{x}}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i} \right) \mathbf{d}_i = \sum_{i=1}^{n} \mathbf{p}_{\mathbf{d}_i}(\mathbf{x}) \tag{A.51}$$

A set of independent basis vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$ can now also converted to an orthogonal basis with respect to a symmetric vector $\mathbf{A}$ by Gram-Schmidt process:

$$\mathbf{d}_k = \mathbf{v}_k - \sum_{i=1}^{k-1} \mathbf{p}_{\mathbf{d}_i}(\mathbf{v}_k) = \mathbf{v}_k - \sum_{i=1}^{k-1} \left(\frac{\mathbf{v}_k^T \mathbf{A}\mathbf{d}_i}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i} \right) \mathbf{d}_i, \qquad (k = 1, \cdots, n) \tag{A.52}$$

A.2 Matrices

A.2.1 Rank, Trace, Determinant, Transpose, and Inverse

Let $\mathbf{A}$ be an $m \times n$ square matrix:

$$\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_n] = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}_{m \times n} \tag{A.53}$$

where

$$\mathbf{a}_j = \begin{bmatrix} a_{1j} \\ \vdots \\ a_{mj} \end{bmatrix}, \qquad (j = 1, \cdots, n) \tag{A.54}$$

is the jth column vector and $[a_{i1} \ \cdots \ a_{in}]$ is the ith row vector $(i = 1, \cdots, m)$. If $m = n$, $\mathbf{A}$ is a *square matrix*. In particular, if all entries of a square matrix are zero except those along the diagonal, it is a *diagonal matrix*. Moreover, if the diagonal entries of a diagonal matrix are all one, it is the *identity matrix*:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} \tag{A.55}$$

- **Rank**

  The m row vectors span the *row space* of $\mathbf{A}$ and the n columns vectors span the *column space* of $\mathbf{A}$. The *rank* of each space is its dimension, the number of independent vectors in the space. The row and column spaces have the same rank, which is also the rank of matrix $\mathbf{A}$, i.e.:

  $$r = \text{rank}(\mathbf{A}) \leq \min(m, n) \tag{A.56}$$

  In other words, the rank of matrix $\mathbf{A}$ is the number of its independent rows or columns.

- **Transpose**

  The *transpose* $\mathbf{x}^T$ a column vector $\mathbf{x}$ is a row vector:

$$\mathbf{x}^T = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}^T = [x_1, \cdots, x_n] \tag{A.57}$$

  The transpose $\mathbf{A}^T$ of a matrix $\mathbf{A}$ is obtained by switching the positions of elements a_{ij} and a_{ji} for all $i \in \{1, \cdots, m\}$ and $j \in \{1, \cdots, n\}$. In other words, the ith column of $\mathbf{A}$ becomes the ith row of $\mathbf{A}^T$, or equivalently, the ith row of $\mathbf{A}$ becomes the ith column of $\mathbf{A}^T$:

$$\mathbf{A}^T = [\mathbf{a}_1 \cdots \mathbf{a}_n]^T = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} \tag{A.58}$$

  where vector $\mathbf{a}_i$ is the ith column of $\mathbf{A}$ and its transpose $\mathbf{a}_i^T$ is the ith row of $\mathbf{A}^T$.

  The properties of the transpose:
  - $(\mathbf{A}^T)^T = \mathbf{A}$
  - $(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$
  - $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$
  - $\mathbf{A}^T$ and $\mathbf{A}$ have the same eigenvalues and eigenvectors.

  If $\mathbf{A} = \mathbf{A}^T$, it is a *symmetric matrix*.

- **Conjugate transpose**

  The *conjugate transpose* of a matrix $\mathbf{A}$, denoted by $\mathbf{A}^*$, is by taking the complex conjugate of the its transpose:

$$\mathbf{A}^* = (\overline{\mathbf{A}})^T = \overline{\mathbf{A}^T} \tag{A.59}$$

  i.e., $(\mathbf{A}^*)_{ij} = \overline{\mathbf{A}}_{ji}$, the ij-th entry of $\mathbf{A}^*$ is the complex conjugate of the ji-th entry of $\mathbf{A}$.

  The properties of the conjugate transpose:
  - $(\mathbf{A}^*)^* = \mathbf{A}$
  - $(c\mathbf{A})^* = \overline{c}\mathbf{A}^*$
  - $(\mathbf{A} + \mathbf{B})^* = \mathbf{A}^* + \mathbf{B}^*$
  - $(\mathbf{AB})^* = \mathbf{B}^*\mathbf{A}^*$
  - $\det(\mathbf{A}^*) = (\det \mathbf{A})^* = \overline{\det \mathbf{A}}$
  - $tr(\mathbf{A}^*) = (tr\mathbf{A})^* = \overline{tr\mathbf{A}}$
  - $(\mathbf{A}^*)^{-1} = (\mathbf{A}^{-1})^*$.
  - The eigenvalues of $\mathbf{A}^*$ are the complex conjugates of the eigenvalues of $\mathbf{A}$. (But their eigenvectors are not related.)
  - $\langle \mathbf{Ax}, \mathbf{y} \rangle = \langle \mathbf{x}, \mathbf{A}^*\mathbf{y} \rangle$

    Proof:

$$\langle \mathbf{Ax}, \mathbf{y} \rangle = (\mathbf{Ax})^*\mathbf{y} = \mathbf{x}^*\mathbf{A}^*\mathbf{y} = \mathbf{x}^*(\mathbf{A}^*\mathbf{y}) = \langle \mathbf{x}, \mathbf{A}^*\mathbf{y} \rangle \tag{A.60}$$

If $\mathbf{A} = \mathbf{A}^*$, it is a *Hermitian matrix*. A real Hermitian matrix is a symmetric matrix. The inverse of an invertible Hermitian matrix is also Hermitian, i.e., if $\mathbf{A}^* = \mathbf{A}$, then $(\mathbf{A}^{-1})^* = \mathbf{A}^{-1}$.

- **Trace**

  The *trace* of a square matrix A is the sum of its diagonal elements:

$$\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii} \tag{A.61}$$

  The properties of the trace:
  - $\text{tr}(c\mathbf{A}) = c\ \text{tr}(\mathbf{A})$
  - $\text{tr}(\mathbf{A}^T) = \text{tr}(\mathbf{A})$
  - $\text{tr}(\mathbf{A} + \mathbf{B}) = \text{tr}(\mathbf{B} + \mathbf{A})$
  - $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$
  - $\text{tr}(\mathbf{A}^T\mathbf{B}) = \text{tr}(\mathbf{AB}^T)$
  - $\text{tr}(\mathbf{R}^{-1}\mathbf{AR}) = \text{tr}(\mathbf{R}^{-1}(\mathbf{AR})) = \text{tr}((\mathbf{AR})\mathbf{R}^{-1}) = \text{tr}(\mathbf{A})$

- **Determinant**

  The *determinant* of a square matrix A is denoted by $\det(\mathbf{A}) = |\mathbf{A}|$, and $\det(\mathbf{A}) \neq 0$ if and only if it is full rank, i.e., $\text{rank}(\mathbf{A}) = n$.

  The properties of the determinant:
  - $\det(\mathbf{I}) = 1$
  - $\det(\mathbf{A}^T) = \det(\mathbf{A})$
  - $\det(\mathbf{A}^{-1}) = \det(\mathbf{A})^{-1}$
  - $\det(\mathbf{AB}) = \det(\mathbf{A})\ \det(\mathbf{B})$
  - $\det(c\mathbf{A}) = c^n \det(\mathbf{A})$

- **Inverse**

  If $\mathbf{A}$ is an $m \times n$ full rank square matrix with $m = n = r$, then there exists an *inverse matrix* $\mathbf{A}^{-1}$ that satisfies $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. The inverse $\mathbf{A}^{-1}$ does not exist if $\mathbf{A}$ is not square ($m \neq n$) or full rank ($r < m = n$).

  The properties of the inverse:
  - $(\mathbf{A}^{-1})^{-1} = \mathbf{A}$
  - $(\mathbf{A}^T)^{-1} = (\mathbf{A}^{-1})^T$
  - $\det(\mathbf{A}^{-1}) = \det(\mathbf{A})^{-1}$
  - $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$

  For example, the solution of a linear equation system $\mathbf{Ax} = \mathbf{b}$ of m equations and $n = m$ variables can be obtained as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

A.2.2 Normal, Unitary, and Similar Matrices

- **Commutative matrices**

  Two square matrices $\mathbf{A}$ and $\mathbf{B}$ *commute* if $\mathbf{AB} = \mathbf{BA}$.

  Obviously all diagonal matrices commute.

- **Normal matrix**

  A square matrix $\mathbf{A}$ is *normal* if it commutes with its conjugate transpose: $\mathbf{A}^*\mathbf{A} = \mathbf{A}\mathbf{A}^*$. If $\mathbf{A}^* = \mathbf{A}^T$ is real, then $\mathbf{A}^T\mathbf{A} = \mathbf{A}\mathbf{A}^T$.

  Obviously unitary matrices ($\mathbf{A}^* = \mathbf{A}^{-1}$), Hermitian matrices ($\mathbf{A}^* = \mathbf{A}$), and skew-Hermitian matrices ($\mathbf{A}^* = -\mathbf{A}$) are all normal. But there exist normal matrices that are not Hermitian.

- **Unitary matrix**

  $\mathbf{U} = [\mathbf{u}_1, \cdots, \mathbf{u}_n]$ is a *unitary matrix* if its conjugate transpose is equal to its inverse $\mathbf{U}^* = \mathbf{U}^{-1}$, i.e., $\mathbf{U}^*\,\mathbf{U} = \mathbf{I}$. When a unitary matrix $\overline{\mathbf{U}} = \mathbf{U}$ is real, it becomes an *orthogonal matrix*, $\mathbf{U}^T = \mathbf{U}^{-1}$.

  The column (or row) vectors of a unitary matrix $\mathbf{A}$ are *orthonormal*, i.e. they are both orthogonal and normalized:

$$\langle \mathbf{u}_i, \mathbf{u}_j \rangle = \mathbf{u}_i^T \overline{\mathbf{u}}_j = \mathbf{u}_j^* \mathbf{u}_i = \sum_k u_{ik}\, \overline{u}_{jk} = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{A.62}$$

  Let λ and $\mathbf{v}$ be an eigenvalue and the corresponding eigenvector of $\mathbf{U}$, i.e., $\mathbf{U}\mathbf{v} = \lambda\mathbf{v}$, then we have

$$\mathbf{v}^*\mathbf{v} = \mathbf{v}^*\mathbf{U}^*\mathbf{U}\mathbf{v} = (\mathbf{U}\mathbf{v})^*\mathbf{U}\mathbf{v} = \lambda^2\mathbf{v}^*\mathbf{v} \tag{A.63}$$

  We see that $\lambda^2 = 1$, i.e., $|\lambda| = 1$.

  A Hermitian matrix $\mathbf{A}$ can be converted to a diagonal matrix $\mathbf{\Lambda}$ (or diagonalized) by a particular unitary matrix $\mathbf{U}$:

$$\mathbf{U}^*\mathbf{A}\mathbf{U} = \mathbf{\Lambda} = \text{diag}[\lambda_1, \cdots, \lambda_n] \tag{A.64}$$

  where $\mathbf{\Lambda}$ is a diagonal matrix, i.e., all its off diagonal elements are 0.

- **Similar matrices**

  Two matrices $\mathbf{A}$ and $\mathbf{B}$ are similar if they are related by $\mathbf{A} = \mathbf{P}^{-1}\mathbf{B}\mathbf{P}$. Similar matrices have the same determinant, trace, and eigenvalues.

A.2.3 Positive/Negative (Semi)-Definite Matrices

Given a Hermitian matrix $\mathbf{A} = \mathbf{A}^*$ and any non-zero vector $\mathbf{x} \neq \mathbf{0}$, we can construct a quadratic form $\mathbf{x}^*\mathbf{A}\mathbf{x}$. The matrix $\mathbf{A}$ is said to be

- *positive definite* $\mathbf{A} > 0$, if $\mathbf{x}^*\mathbf{A}\mathbf{x} > 0$
- *positive semi-definite* $\mathbf{A} \geq 0$, if $\mathbf{x}^*\mathbf{A}\mathbf{x} \geq 0$
- *negative definite* $\mathbf{A} < 0$, if $\mathbf{x}^*\mathbf{A}\mathbf{x} < 0$
- *negative semi-definite* $\mathbf{A} \leq 0$, if $\mathbf{x}^*\mathbf{A}\mathbf{x} \leq 0$
- *indefinite* if there exists $\mathbf{x}$ and $\mathbf{y}$ such that $\mathbf{x}^T\mathbf{A}\mathbf{x} < 0 < \mathbf{y}^T\mathbf{A}\mathbf{y}$.

For example, consider the covariance matrix of a random vector $\mathbf{x}$

$$\mathbf{\Sigma}_x = E[(\mathbf{x} - \mathbf{m}_x)(\mathbf{x} - \mathbf{m}_x)^*] \tag{A.65}$$

The corresponding quadratic form is

$$\begin{aligned}\mathbf{v}^*\boldsymbol{\Sigma}_x\mathbf{v} &= \mathbf{v}^*E[(\mathbf{x}-\mathbf{m}_x)(\mathbf{x}-\mathbf{m}_x)^*]\mathbf{v} \\ &= E[\mathbf{v}^*(\mathbf{x}-\mathbf{m}_x)(\mathbf{x}-\mathbf{m}_x)^*\mathbf{v}] = E(s^2) \geq 0\end{aligned}$$

where $s = \mathbf{v}^*(\mathbf{x}-\mathbf{m}_x)$ is a scalar. Therefore $\boldsymbol{\Sigma}_x$ is positive semi-definite.

As $\mathbf{A}$ is Hermitian, its eigenvalues $\lambda_1, \cdots, \lambda_n$ are real and its eigenvector matrix is unitary $\mathbf{V}^{-1} = \mathbf{V}^*$, by which it can be diagonalized:

$$\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{V}^*\mathbf{A}\mathbf{V} = \boldsymbol{\Lambda}, \quad \text{i.e.} \quad \mathbf{A} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^* \tag{A.66}$$

where vector $\mathbf{y} = \mathbf{V}^*\mathbf{x}$ is a unitary transform of vector $\mathbf{x}$.

For any $\mathbf{x} = [x_1, \cdots, x_n]^T$, we have

$$\mathbf{x}^*\mathbf{A}\mathbf{x} = \mathbf{x}^*(\mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^*)\mathbf{x} = (\mathbf{V}^*\mathbf{x})^*\boldsymbol{\Lambda}(\mathbf{V}^*\mathbf{x}) = \mathbf{y}^*\boldsymbol{\Lambda}\mathbf{y} = \sum_{i=1}^{n}\lambda_i|y_i|^2$$

$$= \begin{cases} > 0 \text{ is positive definite} & \text{if } \lambda_i > 0 \\ \geq 0 \text{ is positive semi-definite} & \text{if } \lambda_i \geq 0 \\ < 0 \text{ is negative definite} & \text{if } \lambda_i < 0 \\ \leq 0 \text{ is negative semi-definite} & \text{if } \lambda_i \leq 0 \end{cases} \quad (\text{for all } i)$$

Also, if some eigenvalues are positive and some others are negative, $\mathbf{x}^*\mathbf{A}\mathbf{x}$ may be either positive or negative depending on $\mathbf{x}$, i.e., $\mathbf{A}$ is indefinite.

As the eigenvalues of $\mathbf{A}^{-1}$ are $1/\lambda_i,\ i = (1, \cdots, n)$, $\mathbf{A}^{-1}$ and $\mathbf{A}$ share the same positive/negative definiteness.

A.2.4 Woodbury Matrix Identity and Sherman-Morrison Formula

The Woodbury matrix identity gives the inverse of an $n \times n$ square matrix $\mathbf{A}$ modified by a perturbation term $\mathbf{U}\mathbf{B}\mathbf{V}^T$

$$(\mathbf{A}+\mathbf{U}\mathbf{B}\mathbf{V}^T)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1} \tag{A.67}$$

Proof:

$$\begin{aligned}&(\mathbf{A}+\mathbf{U}\mathbf{B}\mathbf{V}^T)\ \left[\mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1}\right] \\ &= [\mathbf{I} - \mathbf{U}(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1}] \\ &+ [\mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1} - \mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U}(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1}] \\ &= \mathbf{I} + \mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1} - [\mathbf{U}(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1} + \mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U}(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}\mathbf{A}^{-1}] \\ &= \mathbf{I} + \mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1} - (\mathbf{U}+\mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1}] \\ &= \mathbf{I} + \mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1} - \mathbf{U}\mathbf{B}(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})(\mathbf{B}^{-1}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1} \\ &= \mathbf{I} + \mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1} - \mathbf{U}\mathbf{B}\mathbf{V}^T\mathbf{A}^{-1} = \mathbf{I}\end{aligned} \tag{A.68}$$

Consider some special cases:

- If $\mathbf{U} = \mathbf{V} = \mathbf{I}$, then we get

$$(\mathbf{A} + \mathbf{B})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}(\mathbf{A}^{-1} + \mathbf{B}^{-1})^{-1}\mathbf{A}^{-1} \tag{A.69}$$

- If $\mathbf{B} = \mathbf{I}$, and let $\mathbf{U} = [\mathbf{u}_1, \cdots, \mathbf{u}_m]$ and $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_m]$, then we get the inverse of a rank-n modified matrix:

$$(\mathbf{A}+\mathbf{U}\mathbf{V}^T)^{-1} = \left(\mathbf{A} + \sum_{i=1}^{m} \mathbf{u}_i \mathbf{v}_i^T\right)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I}+\mathbf{V}^T\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^T\mathbf{A}^{-1} \tag{A.70}$$

- More specially, if $m = 1$, $\mathbf{U} = \mathbf{u}$ and $\mathbf{V} = \mathbf{v}$ become vectors, then we get the Sherman-Morrison formula:

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T\mathbf{A}^{-1}}{1 + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}} \tag{A.71}$$

A.2.5 Inverse and Determinant of Partitioned Symmetric Matrices

- **Theorem 1** (inverse of a partitioned symmetric matrix)
  Divide an $n \times n$ symmetric matrix $\mathbf{A}$ into four blocks

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \tag{A.72}$$

  The inverse matrix $\mathbf{B} = \mathbf{A}^{-1}$ can also be divided into four blocks:

$$\mathbf{B} = \mathbf{A}^{-1} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} \tag{A.73}$$

  Here we assume the dimensionalities of these blocks are:
  – $\mathbf{A}_{11}$ and $\mathbf{B}_{11}$ are $p \times p$,
  – $\mathbf{A}_{22}$ and $\mathbf{B}_{22}$ are $q \times q$,
  – $\mathbf{A}_{12} = \mathbf{A}_{21}^T$ and $\mathbf{B}_{12} = \mathbf{B}_{21}^T$ are $p \times q$
  with $p + q = n$. Then we have

$$\begin{aligned}
\mathbf{B}_{11} &= (\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21})^{-1} = \mathbf{A}_{11}^{-1} + \mathbf{A}_{11}^{-1}\mathbf{A}_{12}(\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})^{-1}\mathbf{A}_{21}\mathbf{A}_{11}^{-1} \\
\mathbf{B}_{22} &= (\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})^{-1} = \mathbf{A}_{22}^{-1} + \mathbf{A}_{22}^{-1}\mathbf{A}_{21}(\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21})^{-1}\mathbf{A}_{12}\mathbf{A}_{22}^{-1} \\
\mathbf{B}_{21} &= -\mathbf{A}_{22}^{-1}\mathbf{A}_{21}(\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21})^{-1} \\
\mathbf{B}_{12} &= -\mathbf{A}_{11}^{-1}\mathbf{A}_{12}(\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})^{-1}
\end{aligned} \tag{A.74}$$

  Proof:

$$\begin{aligned}
\mathbf{I}_n = \mathbf{A}\mathbf{A}^{-1} = \mathbf{A}\mathbf{B} &= \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_p & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_q \end{bmatrix}
\end{aligned}$$

Equate each of the four blocks to get:

$$\begin{array}{lll} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} = \mathbf{I}_p, & \text{or} & \mathbf{B}_{11} = \mathbf{A}_{11}^{-1} - \mathbf{A}_{11}^{-1}\mathbf{A}_{12}\mathbf{B}_{21} \\ \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} = \mathbf{0}, & \text{or} & \mathbf{B}_{12} = -\mathbf{A}_{11}^{-1}\mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} = \mathbf{0}, & \text{or} & \mathbf{B}_{21} = -\mathbf{A}_{22}^{-1}\mathbf{A}_{21}\mathbf{B}_{11} \\ \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} = \mathbf{I}_q & \text{or} & \mathbf{B}_{22} = \mathbf{A}_{22}^{-1} - \mathbf{A}_{22}^{-1}\mathbf{A}_{21}\mathbf{B}_{12} \end{array} \tag{A.75}$$

Plug $\mathbf{B}_{21}$ into $\mathbf{B}_{11}$ to get:

$$\mathbf{B}_{11} = \mathbf{A}_{11}^{-1} + \mathbf{A}_{11}^{-1}\mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21}\mathbf{B}_{11} \tag{A.76}$$

Solve for $\mathbf{B}_{11}$ to get:

$$\mathbf{B}_{11} = (\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21})^{-1} \tag{A.77}$$

Applying theorem 1 to this expression, we also get the other expression in the theorem. Similarly we can get

$$\begin{aligned} \mathbf{B}_{22} &= (\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})^{-1} \\ \mathbf{B}_{21} &= -\mathbf{A}_{22}^{-1}\mathbf{A}_{21}(\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21})^{-1} \\ \mathbf{B}_{12} &= -\mathbf{A}_{11}^{-1}\mathbf{A}_{12}(\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}^{-1})^{-1} \end{aligned}$$

- **Theorem 2** (Determinant of a partitioned symmetric matrix)

$$|\mathbf{A}| = \left|\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}\right| = |\mathbf{A}_{22}|\,|\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21}| = |\mathbf{A}_{11}|\,|\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}| \tag{A.78}$$

Proof:

$$\begin{aligned} \mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} &= \begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{A}_{11}^{-1}\mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21} & \mathbf{0} \\ \mathbf{A}_{22}^{-1}\mathbf{A}_{21} & \mathbf{I} \end{bmatrix} \end{aligned}$$

The theorem is proved as we also know that

$$|\mathbf{BC}| = |\mathbf{B}|\,|\mathbf{C}| \tag{A.79}$$

and

$$\begin{vmatrix} \mathbf{B} & \mathbf{0} \\ \mathbf{X} & \mathbf{C} \end{vmatrix} = \begin{vmatrix} \mathbf{B} & \mathbf{X} \\ \mathbf{0} & \mathbf{C} \end{vmatrix} = |\mathbf{B}|\,|\mathbf{C}| \tag{A.80}$$

A.2.6 Unitary Transform

Given any *unitary matrix* $\mathbf{A}$ satisfying $\mathbf{A}\mathbf{A}^* = \mathbf{A}^*\mathbf{A} = \mathbf{I}$, we can define a *unitary transform* of a vector $\mathbf{x} = [x_1, \cdots, x_n]^T$:

$$\begin{cases} \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \mathbf{A}^*\mathbf{x} = \begin{bmatrix} \overline{\mathbf{a}}_1^T \\ \vdots \\ \overline{\mathbf{a}}_n^T \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, & \text{(forward transform)} \\ \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \mathbf{A}\mathbf{y} = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^n y_i\, \mathbf{a}_i & \text{(inverse transform)} \end{cases} \tag{A.81}$$

When $\mathbf{A} = \overline{\mathbf{A}}$ is real, $\mathbf{A}^{-1} = \mathbf{A}^T$ is an orthogonal matrix and the corresponding transform is an orthogonal transform.

The first equation above is the *forward transform* and can be written in component form as:

$$y_i = \overline{\mathbf{a}}_i^T\mathbf{x} = \langle \mathbf{x}, \mathbf{a}_i \rangle = \sum_{j=1}^n x_j \overline{a}_{ij}, \qquad (i = 1, \cdots, n) \tag{A.82}$$

The transform coefficient is an inner product $y_i = \langle \mathbf{x}, \mathbf{a}_i \rangle$, representing the projection of vector $\mathbf{x}$ onto the ith column vector $\mathbf{a}_i$ of the transform matrix $\mathbf{A}$. The second equation is the *inverse transform* and can also be written in component form as:

$$x_j = \sum_{i=1}^n a_{ji}\, y_i, \qquad (j = 1, \cdots, n) \tag{A.83}$$

By this transform, vector $\mathbf{x}$ is represented as a linear combination (weighted sum) of the n column vectors $\mathbf{a}_i, \mathbf{a}_2, \cdots, \mathbf{a}_n$ of matrix $\mathbf{A}$. Geometrically, $\mathbf{x}$ is a point in the n-dimensional space spanned by these n orthonormal basis vectors. Each coefficient (coordinate) y_i is the projection of $\mathbf{x}$ onto the corresponding basis vector $\mathbf{a}_i$.

As the n-dimensional space can be spanned by the column vectors of *any* $n \times n$ unitary (orthogonal) matrix, a vector $\mathbf{x}$ in the space can be represented by any of such matrices, each defining a different transform.

Example A.1 When $\mathbf{A} = \mathbf{I} = [\cdots, \mathbf{e}_i, \cdots]$ is an identity matrix, we have

$$\mathbf{x} = \sum_{i=1}^n y_i \mathbf{a}_i = \sum_{i=1}^n x_i \mathbf{e}_i \tag{A.84}$$

where $\mathbf{e}_i = [0, \cdots, 0, 1, 0, \cdots, 0]^T$ is the ith column of $\mathbf{I}$ with the ith element equal 1 and all other 0.

Example A.2 When $a_{m,n} = w[m, n] = e^{-j2\pi mn/N}$, the corresponding transform is discrete Fourier transform. The nth column vector $\mathbf{w}_n$ of the transform

matrix $\mathbf{W} = [\mathbf{w}_0, \cdots, \mathbf{w}_{N-1}]$ represents a sinusoid of a frequency nf_0, and the corresponding complex coordinate $y_n = (\mathbf{x}, \mathbf{w}_n)$ represents the magnitude $|y_n|$ and phase $\angle y_n$ of this nth frequency component. The Fourier transform $\mathbf{y} = \mathbf{W}\mathbf{x}$ represents a rotation of the coordinate system.

A unitary (orthogonal if real) transform $\mathbf{y} = \mathbf{A}\,\mathbf{x}$ can be interpreted geometrically as the rotation of the vector $\mathbf{x}$ about the origin, or equivalently, the representation of the same vector in a rotated coordinate system. A unitary transform $\mathbf{y} = \mathbf{A}\,\mathbf{x}$ does not change the inner product. Let $\mathbf{u} = \mathbf{A}^*\mathbf{x}$ and $\mathbf{v} = \mathbf{A}^*\mathbf{y}$ be the unitary transforms of vectors $\mathbf{x}$ and $\mathbf{t}$, then the inner product of these two vectors is preserved:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{A}^*\mathbf{u}, \mathbf{A}^*\mathbf{v} \rangle = \mathbf{x}^*\mathbf{A}\mathbf{A}^*\mathbf{y} = \mathbf{x}^*\mathbf{y} = \langle \mathbf{x}, \mathbf{y} \rangle \tag{A.85}$$

In particular, if $\mathbf{x} = \mathbf{y}$, we see that the vector norm is preserved by the unitary transform:

$$||\mathbf{u}||^2 = \langle \mathbf{u}, \mathbf{u} \rangle = \langle \mathbf{x}, \mathbf{x} \rangle = ||\mathbf{x}||^2 \tag{A.86}$$

This is the Parseval's identity that indicates that the norm (length) of a vector is preserved under any unitary transform. If $\mathbf{X}$ is interpreted as a signal, then its length $||\mathbf{x}||^2 = ||\mathbf{y}||^2$ represents the total energy or information contained in the signal, which is conserved during any unitary transform. However, some other features of the signal may change, e.g., the signal may be decorrelated and its total energy redistributed among its components after the transform, which may be desirable in many applications.

If $\mathbf{x}$ is a random vector with mean vector $\mathbf{m}_x$ and covariance matrix $\mathbf{\Sigma}_x$:

$$\mathbf{m}_x = E(\mathbf{x}), \qquad \mathbf{\Sigma}_x = E(\mathbf{x}\mathbf{x}^*) - \mathbf{m}_x\mathbf{m}_x^* \tag{A.87}$$

then its transform $\mathbf{y} = \mathbf{A}^*\mathbf{x}$ has the following mean vector and covariance matrix:

$$\mathbf{m}_y = E(\mathbf{y}) = E(\mathbf{A}^*\mathbf{x}) = \mathbf{A}^*E(\mathbf{x}) = \mathbf{A}^*\mathbf{m}_x \tag{A.88}$$

$$\begin{aligned} \mathbf{\Sigma}_y &= E(\mathbf{y}\mathbf{y}^*) - \mathbf{m}_y\mathbf{m}_y^* = E[(\mathbf{A}^*\mathbf{x})(\mathbf{A}^*\mathbf{x})^*] - (\mathbf{A}^*\mathbf{m}_x)(\mathbf{A}^*\mathbf{m}_x)^* \\ &= E[\mathbf{A}^*(\mathbf{x}\mathbf{x}^*)\mathbf{A}] - \mathbf{A}^*\mathbf{m}_x\mathbf{m}_x^*\mathbf{A} = \mathbf{A}^*[E(\mathbf{x}\mathbf{x}^*) - \mathbf{m}_x\mathbf{m}_x^*]\mathbf{A} \\ &= \mathbf{A}^*\mathbf{\Sigma}_x\mathbf{A} \end{aligned} \tag{A.89}$$

In general the unitary transform of any square matrix $\mathbf{A}$ by a unitary matrix $\mathbf{R}$ is $\mathbf{B} = \mathbf{R}^*\mathbf{A}\mathbf{R}$.

A.3 Eigenvalues and Eigenvectors

A.3.1 Eigenvalue Decomposition

For any $n \times n$ square matrix $\mathbf{A}$, if there exist a vector $\mathbf{v}$ and a scalar λ such that the following *eigenequation* holds:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}, \tag{A.90}$$

then λ and $\mathbf{v}$ are called the *eigenvalue* and *eigenvector* of matrix $\mathbf{A}$, respectively. In other words, the linear transformation of vector $\mathbf{v}$ by $\mathbf{A}$ has the same effect of scaling the vector by factor λ. (Note that for an $m \times n$ non-square matrix $\mathbf{A}$ with $m \neq n$, $\mathbf{A}\mathbf{v}$ is an m-D vector but $\lambda\mathbf{v}$ is n-D vector, i.e., no eigenvalues and eigenvectors are defined.)

Given $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$, we also have $\mathbf{A}c\mathbf{v} = \lambda c\mathbf{v}$ for any scalar constant c, i.e., the eigenvector $\mathbf{v}$ is not unique but up to any scaling factor. For the uniqueness of $\mathbf{v}$, we typically keep it normalized so that $||\mathbf{v}|| = 1$.

To obtain λ, we rewrite the above equation as

$$(\lambda\mathbf{I} - \mathbf{A})\mathbf{v} = \mathbf{0} \tag{A.91}$$

For this homogeneous equation system to have non-zero solutions for $\mathbf{v}$, the determinant of its coefficient matrix has to be zero:

$$\det(\lambda\mathbf{I} - \mathbf{A}) = |\lambda\mathbf{I} - \mathbf{A}| = 0 \tag{A.92}$$

This is the *characteristic polynomial equation* of matrix $\mathbf{A}$. Solving this nth order equation of λ, we get n eigenvalues $\{\lambda_1, \cdots, \lambda_n\}$. Substituting each λ_i back into the homogeneous equation system, we get the corresponding eigenvector $\mathbf{v}_i$. We can put all n eigenequations $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$ together and obtain the more compact form:

$$\begin{aligned}\mathbf{A}\mathbf{V} = \mathbf{A}[\mathbf{v_1}, \cdots, \mathbf{v}_n] &= [\lambda_1\mathbf{v}_1, \cdots, \lambda_n\mathbf{v}_n] \\ &= [\mathbf{v}_1, \cdots, \mathbf{v}_n]\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} = \mathbf{V}\mathbf{\Lambda}\end{aligned} \tag{A.93}$$

where we have defined

$$\mathbf{\Lambda} = \text{diag}[\lambda_1, \cdots, \lambda_n] \quad \text{and} \quad \mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_n] \tag{A.94}$$

The eigenequation can be written in some alternative forms:

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}, \quad \text{or} \quad \mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{\Lambda} \tag{A.95}$$

In the first form, $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$ is expressed as a product of three matrices, called *eigenvalue decomposition* of the matrix; in the second form, $\mathbf{A}$ is diagonalized by its eigenvector matrix $\mathbf{V}$ to become a diagonal matrix, its eigenvalue matrix $\mathbf{\Lambda}$.

Given $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$, we have the following:

- $\mathbf{A}^T$ has the same eigenvalues and eigenvectors as $\mathbf{A}$.
  Proof: As a matrix $\mathbf{A}$ and its transpose $\mathbf{A}^T$ have the same determinant, they have the same characteristic polynomial:
  $$\left|\mathbf{A} - \lambda\mathbf{I}\right| = \left|(\mathbf{A} - \lambda\mathbf{I})^T\right| = \left|\mathbf{A}^T - \lambda\mathbf{I}\right| \tag{A.96}$$
  therefore they have the same eigenvalues and eigenvectors.
- The eigenvalues and eigenvectors of $\mathbf{A}^*$ are the complex conjugate of the eigenvalues and eigenvectors as $\mathbf{A}$.
- $\mathbf{A}^T\mathbf{A}$ has the same eigenvectors as $\mathbf{A}$, but its eigenvalues are λ^2.
  Proof:
  $$\mathbf{A}^T\mathbf{A}\mathbf{v} = \mathbf{A}^T\lambda\mathbf{v} = \lambda^2\mathbf{v} \tag{A.97}$$
- $\mathbf{A}^k$ has the same eigenvectors as $\mathbf{A}$, but its eigenvalues are $\{\lambda_1^k, \cdots, \lambda_n^k\}$, where k is a positive integer.
  Proof:
  $$\mathbf{A}^2\mathbf{v} = \mathbf{A}\mathbf{A}\mathbf{v} = \mathbf{A}\lambda\mathbf{v} = \lambda^2\mathbf{v} \tag{A.98}$$
  This result can be generalized to
  $$\mathbf{A}^k\mathbf{v} = \lambda^k\mathbf{v} \tag{A.99}$$
- In particular when $k = -1$, i.e., the eigenvalues of $\mathbf{A}^{-1}$ are $\{\lambda_1^{-1}, \cdots, \lambda_n^{-1}\}$.
  Proof: Left multiplying $\mathbf{A}^{-1}$ on both sides of $\mathbf{A}\mathbf{v}_i = \lambda\mathbf{v}_i$ we get
  $$\mathbf{A}^{-1}\mathbf{A}\mathbf{v}_i = \mathbf{I}\mathbf{v}_i = \mathbf{v}_i = \lambda_i\mathbf{A}^{-1}\mathbf{v}_i \tag{A.100}$$
  Dividing both sides by λ we get
  $$\mathbf{A}^{-1}\mathbf{v}_i = \frac{1}{\lambda_i}\mathbf{v}_i = \lambda_i^{-1}\mathbf{v}_i \tag{A.101}$$
- The eigenvalues of a matrix are invariant under any unitary transform $\mathbf{B} = \mathbf{R}^*\mathbf{A}\mathbf{R}$, where $\mathbf{R}$ is unitary, satisfying $\mathbf{R}^{-1} = \mathbf{R}^*$, or $\mathbf{R}^*\mathbf{R} = \mathbf{R}\mathbf{R}^* = \mathbf{I}$
  Proof:
  Let $\mathbf{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_n)$ and $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_n]$ be the eigenvalue and eigenvector matrices of a square matrix $\mathbf{A}$:
  $$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Lambda} \tag{A.102}$$
  and $\mathbf{\Sigma} = \text{diag}(\sigma_1, \cdots, \sigma_n)$ and $\mathbf{U} = [\mathbf{u}_1, \cdots, \mathbf{u}_n]$ be the eigenvalue and eigenvector matrices of $\mathbf{B} = \mathbf{R}^*\mathbf{A}\mathbf{R}$, a unitary transform of $\mathbf{A}$:
  $$\mathbf{B}\mathbf{U} = (\mathbf{R}^*\mathbf{A}\mathbf{R})\,\mathbf{U} = \mathbf{U}\mathbf{\Sigma} \tag{A.103}$$
  Left-multiplying $\mathbf{R}$ on both sides we get the eigenequation of $\mathbf{A}$
  $$\mathbf{R}\mathbf{R}^*\mathbf{A}\mathbf{R}\mathbf{U} = \mathbf{A}(\mathbf{R}\mathbf{U}) = (\mathbf{R}\mathbf{U})\mathbf{\Sigma} \tag{A.104}$$
  We see that $\mathbf{A}$ and $\mathbf{B} = \mathbf{R}^*\mathbf{A}\mathbf{R}$ have the same eigenvalues $\mathbf{\Sigma} = \mathbf{\Lambda}$ and their eigenvector matrices are related by $\mathbf{V} = \mathbf{R}\mathbf{U}$ or $\mathbf{U} = \mathbf{R}^*\mathbf{V}$.

- Given all eigenvalues $\lambda_1, \cdots, \lambda_n$ of a matrix $\mathbf{A}$, its trace and determinant can be obtained as

$$\text{tr}(\mathbf{A}) = \sum_{k=1}^{n} \lambda_k, \qquad \det(\mathbf{A}) = \prod_{k=1}^{n} \lambda_k \tag{A.105}$$

- The *spectrum* of an $n \times n$ square matrix $\mathbf{A}$ is the set of its eigenvalues $\{\lambda_1, \cdots, \lambda_n\}$. The *spectral radius* of $\mathbf{A}$, denoted by $\rho(\mathbf{A})$, is the maximum of the absolute values of the elements in the spectrum:

$$\rho(\mathbf{A}) = \max(|\lambda_1|, \cdots, |\lambda_n|) \tag{A.106}$$

where $|z| = \sqrt{x^2 + y^2}$ is the modulus of a complex number $z = x + jy$. If all eigenvalues are sorted such that $|\lambda_1| \geq \cdots \geq |\lambda_n|$ then $\rho(\mathbf{A}) = |\lambda_1| = |\lambda_{max}|$. As the eigenvalues of $\mathbf{A}^{-1}$ are $\{1/\lambda_{max}, \cdots, 1/\lambda_{min}\}$, $\rho(\mathbf{A}^{-1}) = 1/|\lambda_{min}$.

- If $\mathbf{A}$ and $\mathbf{B}$ are similar, i.e.,

$$\mathbf{B} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P} \tag{A.107}$$

then they have the same eigenvalues.

Proof: Let $\mathbf{\Lambda}$ and $\mathbf{V}$ be the eigenvalue and eigenvector matrices of $\mathbf{A}$:

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Lambda}, \qquad \mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} \tag{A.108}$$

then we have

$$\mathbf{B} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P} = \mathbf{P}^{-1}\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}\mathbf{P} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1} \tag{A.109}$$

i.e., $\mathbf{\Lambda}$ and $\mathbf{U} = \mathbf{P}^{-1}\mathbf{V}$ are the eigenvalue and eigenvector matrices of $\mathbf{B} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P}$.

- If $\mathbf{A}$ is Hermitian $\mathbf{A}^* = \bar{\mathbf{A}}^T = \mathbf{A}$ (symmetric $\mathbf{A}^T = \mathbf{A}$ if real) (e.g., the covariance matrix of a random vector), then all of its eigenvalues $\bar{\lambda}_i = \lambda_i$ are real, and all of its eigenvectors are orthogonal, $\mathbf{v}_i^*\mathbf{v}_j = 0$ $(i \neq j)$ i.e., $\mathbf{V}^{-1} = \mathbf{V}^*$. And we further have

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T = \mathbf{V}\mathbf{\Lambda}^{1/2}\mathbf{\Lambda}^{1/2}\mathbf{V}^T = \mathbf{U}\mathbf{U}^T \tag{A.110}$$

where $\mathbf{U} = \mathbf{V}\mathbf{\Lambda}^{1/2}$.

Proof:

Let λ and $\mathbf{v}$ be an eigenvalue and the corresponding eigenvector of a Hermitian matrix $\mathbf{A} = \mathbf{A}^*$, i.e., $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$, then we have

$$\begin{aligned}(\mathbf{A}\mathbf{v})^*\mathbf{v} &= (\lambda\mathbf{v})^*\mathbf{v} = \bar{\lambda}\mathbf{v}^*\mathbf{v} = \bar{\lambda}\,||\mathbf{v}||^2 \\ &= \mathbf{v}^*\mathbf{A}^*\mathbf{v} = \mathbf{v}^*\mathbf{A}\mathbf{v} = \mathbf{v}^*\lambda\mathbf{v} = \lambda\,||\mathbf{v}||^2\end{aligned} \tag{A.111}$$

i.e., $\bar{\lambda} = \lambda$ is real. We also have

$$\begin{aligned}&\mathbf{v}_i^*\mathbf{A}\mathbf{v}_j = \mathbf{v}_i^*\lambda_j\mathbf{v}_j = \lambda_j\mathbf{v}_i^*\mathbf{v}_j \\ &= (\mathbf{v}_j^*\mathbf{A}\mathbf{v}_i)^* = (\mathbf{v}_j^*\lambda_i\mathbf{v}_i)^* = \bar{\lambda}_i\mathbf{v}_i^*\mathbf{v}_j = \lambda_i\mathbf{v}_i^*\mathbf{v}_j\end{aligned} \tag{A.112}$$

i.e.,

$$\lambda_j \mathbf{v}_i^* \mathbf{v}_j = \lambda_i \mathbf{v}_i^* \mathbf{v}_j, \quad \text{or} \quad (\lambda_i - \lambda_j)\, \mathbf{v}_i^* \mathbf{v}_j = 0 \tag{A.113}$$

As $\lambda_i \neq \lambda_j$, we get $\mathbf{v}_i^* \mathbf{v}_j = 0$, i.e., the eigenvectors corresponding to different eigenvalues are orthogonal.

QED

When all eigenvectors are normalized $\mathbf{v}_i^* \mathbf{v}_i = ||\mathbf{v}_i||^2 = 1$, they become orthonormal

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle = \mathbf{v}_i^* \mathbf{v}_j = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{A.114}$$

i.e., the eigenvector matrix $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_n]^T$ is unitary (orthogonal if $\mathbf{A}$ is real):

$$\mathbf{V}^{-1} = \mathbf{V}^* \quad \text{i.e.} \quad \mathbf{V}^* \mathbf{V} = \mathbf{I} \tag{A.115}$$

and we have

$$\mathbf{V}^{-1} \mathbf{A} \mathbf{V} = \mathbf{V}^* \mathbf{A} \mathbf{V} = \mathbf{\Lambda}, \tag{A.116}$$

Left and right multiplying by $\mathbf{V}$ and $\mathbf{V}^* = \mathbf{V}^{-1}$ respectively on the two sides, we get

$$\mathbf{A} = \mathbf{V} \mathbf{\Lambda} \mathbf{V}^* = [\mathbf{v}_1, \cdots, \mathbf{v}_n] \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^* \\ \vdots \\ \mathbf{v}_n^* \end{bmatrix} = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^* \tag{A.117}$$

This is the *spectral theorem* indicating that $\mathbf{A}$ can be written as a linear combination of n matrices $\mathbf{v}_i \mathbf{v}_i^*$ weighted by λ_i $(n = 1, \cdots, n)$.

The significance of this property is that a linear operation $\mathbf{y} = \mathbf{A}\mathbf{x}$ applied to vector $\mathbf{x}$ can be mapped to a new vector space in which the operations of the components are independent of each other. Consider

$$\mathbf{y} = \mathbf{A}\mathbf{x} = \mathbf{V} \mathbf{\Lambda} \mathbf{V}^* \mathbf{x} \tag{A.118}$$

Pre-multiplying $\mathbf{V}^*$ on both sides we get

$$\mathbf{y}' = \mathbf{V} * \mathbf{y} = \mathbf{\Lambda} \mathbf{V}^* \mathbf{x} = \mathbf{\Lambda} \mathbf{x}' \tag{A.119}$$

where we have defined $\mathbf{x}' = \mathbf{V}^* \mathbf{x}$ and $\mathbf{y}' = \mathbf{V}^* \mathbf{y}$, the unitary transform of $\mathbf{x}$ and $\mathbf{y}$, respectively. We see that for the ith component we have the following, independent of all other components:

$$y_i' = \lambda_i x_i' \tag{A.120}$$

- The entries on the diagonal of an upper (or lower) triangular matrix are its eigenvalues.

  Proof: Let $\mathbf{A}$ be an upper triangular matrix with $a_{ij} = 0$ for all $i > j$.

The eigenvalues of $\mathbf{A}$ are the roots of the following homogeneous characteristic equations:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = \prod_{i=1}^{n}(a_{ii} - \lambda) = 0 \tag{A.121}$$

The first equal sign is due to the fact that $\mathbf{A}-\lambda\mathbf{I}$ is also an upper-triangular matrix, and the determinant of an upper-triangular matrix is the product of all its diagonal entries. We therefore see that each diagonal entry a_{ii}, as a root of the characteristic equation, is also an eigenvalue of $\mathbf{A}$.

QED

- Similar matrices have the same eigenvalues.

  Proof: Let λ and $\mathbf{v}$ be an eigenvalue and the corresponding eigenvector of $\mathbf{A}$ satisfying $\mathbf{Av} = \lambda\mathbf{v}$, and $\mathbf{B} = \mathbf{P}^{-1}\mathbf{AP}$ be a similar matrix of $\mathbf{A}$. We have $\mathbf{A} = \mathbf{PBP}^{-1}$ and

$$\mathbf{Av} = \mathbf{PBP}^{-1}\mathbf{v} = \lambda\mathbf{v}, \qquad \text{i.e.,} \qquad \mathbf{BP}^{-1}\mathbf{v} = \lambda\mathbf{P}^{-1}\mathbf{v} \tag{A.122}$$

  In other words, λ is also an eigenvalue of $\mathbf{B}$ with the corresponding eigenvector $\mathbf{P}^{-1}\mathbf{v}$.

  QED

- All eigenvalues of a *stochastic matrix* $\mathbf{P}$ are no greater than 1. A stochastic matrix is a matrix of which component P_{ij} is the probability for a state transition of a Markov process (chain) from s_i to s_j, i.e., $0 \le P_{ij} \le 1$ and $\sum_j P_{ij} = 1$.

  Proof: First, as $\mathbf{P1} = \mathbf{1}$, $\lambda = 1$ is one of the eigenvalues of $\mathbf{P}$. Next let λ and $\mathbf{v}$ be an eigenvalue and the corresponding eigenvector of $\mathbf{P}$, i.e., $\mathbf{Pv} = \lambda\mathbf{v}$, and let $|v_k| \ge \max\{|v_1|, \cdots, |v_n|\}$. The kth row of the eigenequation is

$$|\lambda v_k| \le |\lambda|\ |v_k| = \left|\sum_{j=1}^{n} P_{kj}v_j\right| \le \sum_{j=1}^{n} P_{kj}\ |v_j| \le |v_k| \sum_{j=1}^{n} P_{kj} = |v_k| \tag{A.123}$$

  i.e., $|\lambda| \le 1$.

  QED

A.3.2 Generalized Eigenvalue Problem

The *generalized eigenvalue problem* of two symmetric matrices $\mathbf{A} = \mathbf{A}^T$ and $\mathbf{B} = \mathbf{B}^T$ is to find a scalar λ_i and the corresponding vector $\mathbf{v}_i$ for the following generalized eigen equation to hold:

$$\mathbf{Av}_i = \lambda_i\mathbf{Bv}_i, \qquad (i = 1, \cdots, n) \tag{A.124}$$

or in matrix form

$$\mathbf{AV} = \mathbf{BV}\boldsymbol{\Lambda} \tag{A.125}$$

The eigenvalue and eigenvector matrices $\boldsymbol{\Lambda}$ and $\mathbf{V}$ can be found in the following steps.

- Solve the eigenvalue problem of $\mathbf{B}$ to find its diagonal eigenvalue matrix $\mathbf{\Lambda}_B$ and orthogonal eigenvector matrix $\mathbf{V}_B = (\mathbf{V}_B^T)^{-1}$ so that

$$\mathbf{B}\mathbf{V}_B = \mathbf{V}_B\mathbf{\Lambda}_B, \qquad \text{or} \qquad \mathbf{V}_B^{-1}\mathbf{B}\mathbf{V}_B = \mathbf{V}_B^T\mathbf{B}\mathbf{V}_B = \mathbf{\Lambda}_B \tag{A.126}$$

- Left and right multiplying both sides of the second equation above by $\mathbf{\Lambda}^{-1/2}$ (whitening) we get

$$\mathbf{\Lambda}_B^{-1/2}(\mathbf{V}_B^T\mathbf{B}\mathbf{V}_B)\mathbf{\Lambda}_B^{-1/2} = \mathbf{\Lambda}_B^{-1/2}\mathbf{\Lambda}_B\mathbf{\Lambda}_B^{-1/2} = \mathbf{I} \tag{A.127}$$

We define

$$\mathbf{V}'_B = \mathbf{V}_B\mathbf{\Lambda}_B^{-1/2} \tag{A.128}$$

and the above becomes

$$(\mathbf{V}'_B)^T\mathbf{B}\mathbf{V}'_B = \mathbf{I} \tag{A.129}$$

Note that $\mathbf{V}'_B$ is not orthogonal

$$(\mathbf{V}')_B^{-1} = (\mathbf{V}_B\mathbf{\Lambda}_B^{-1/2})^{-1} = \mathbf{\Lambda}_B^{1/2}\mathbf{V}_B^{-1} = \mathbf{\Lambda}_B^{1/2}\mathbf{V}_B^T \neq \mathbf{\Lambda}_B^{-1/2}\mathbf{V}_B^T = \mathbf{V}_B^T \tag{A.130}$$

- Apply the same transform to $\mathbf{A}$:

$$(\mathbf{V}'_B)^T\mathbf{A}\mathbf{V}'_B = (\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_B^T)\mathbf{A}(\mathbf{V}_B\mathbf{\Lambda}_B^{-1/2}) = \mathbf{A}' \tag{A.131}$$

Note that $\mathbf{A}'$ is symmetric as well as $\mathbf{A}$:

$$\mathbf{A}'^T = (\mathbf{V}'^T_B\mathbf{A}\mathbf{V}'_B)^T = \mathbf{V}'^T_B\mathbf{A}\mathbf{V}'_B = \mathbf{A}' \tag{A.132}$$

- Diagonalize $\mathbf{A}'$

As $\mathbf{A}'$ is symmetric,, it can be diagonalized by its orthogonal eigenvector matrix $\mathbf{V}_A$:

$$\mathbf{V}_A^T\mathbf{A}'\mathbf{V}_A = \mathbf{\Lambda} \tag{A.133}$$

i.e.,

$$\begin{aligned}\mathbf{V}_A^T(\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_B^T\mathbf{A}\mathbf{V}_B\mathbf{\Lambda}_B^{-1/2})\mathbf{V}_A &= (\mathbf{V}_A^T\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_B^T)\mathbf{A}(\mathbf{V}_B\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_A)\\ &= \mathbf{V}^T\mathbf{A}\mathbf{V} = \mathbf{\Lambda}\end{aligned}$$

where we have defined

$$\mathbf{V} = \mathbf{V}_B\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_A \tag{A.134}$$

which is not orthogonal:

$$\mathbf{V}^{-1} = (\mathbf{V}_B\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_A)^{-1} = \mathbf{V}_A^{-1}\mathbf{\Lambda}_B^{1/2}\mathbf{V}_B^{-1} = \mathbf{V}_A^T\mathbf{\Lambda}_B^{1/2}\mathbf{V}_B^T \neq \mathbf{V}_A^T\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_B^T = \mathbf{V}^T \tag{A.135}$$

- This $\mathbf{V}$ also diagonalizes $\mathbf{B}$:

$$\begin{aligned}\mathbf{V}^T\mathbf{B}\mathbf{V} &= (\mathbf{V}_B\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_A)^T\mathbf{B}(\mathbf{V}_B\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_A) = \mathbf{V}_A^T\mathbf{\Lambda}_B^{-1/2}(\mathbf{V}_B^T\mathbf{B}\mathbf{V}_B)\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_A\\ &= \mathbf{V}_A^T\mathbf{\Lambda}_B^{-1/2}\mathbf{\Lambda}_B\mathbf{\Lambda}_B^{-1/2}\mathbf{V}_A = \mathbf{V}_A^T\mathbf{V}_A = \mathbf{I}\end{aligned} \tag{A.136}$$

- Now we have

$$\begin{cases} \mathbf{V}^T\mathbf{A}\mathbf{V} = \mathbf{\Lambda} \\ \mathbf{V}^T\mathbf{B}\mathbf{V} = \mathbf{I} \end{cases} \tag{A.137}$$

Right multiplying both sides of the second equation by $\mathbf{\Lambda}$ we get $\mathbf{V}^T\mathbf{B}\mathbf{V}\mathbf{\Lambda} = \mathbf{\Lambda}$, then equating the left-hand side to that of the first equation, we get

$$\mathbf{V}^T\mathbf{A}\mathbf{V} = \mathbf{V}^T\mathbf{B}\mathbf{V}\mathbf{\Lambda}, \qquad \text{i.e.} \qquad \mathbf{A}\mathbf{V} = \mathbf{B}\mathbf{V}\mathbf{\Lambda} \tag{A.138}$$

i.e., $\mathbf{\Lambda}$ and $\mathbf{V}$ are the eigenvalue and eigenvector matrices of the generalized eigenvalue problem. Note, however, as shown above, $\mathbf{V}$ is not orthogonal.

A.3.3 Rayleigh Quotient

The *Rayleigh quotient* $R(\mathbf{A})$ of a hermitian matrix $\mathbf{A} = \mathbf{A}^*$ (symmetric if $\mathbf{A}$ is real) is defined as

$$R(\mathbf{A}) = \frac{\mathbf{w}^*\mathbf{A}\mathbf{w}}{\mathbf{w}^*\mathbf{w}} \tag{A.139}$$

where $\mathbf{w}$ is any non-zero vector. In particular, if $\mathbf{w} = \mathbf{v}_i$ is the ith eigenvector of $\mathbf{A}$, then the Rayleigh quotient is the corresponding to eigenvalue λ_i:

$$R(\mathbf{A}) = \frac{\mathbf{v}_i^*\mathbf{A}\mathbf{v}_i}{\mathbf{v}_i^*\mathbf{v}_i} = \frac{\lambda_i\mathbf{v}_i^*\mathbf{v}_i}{\mathbf{v}_i^*\mathbf{v}_i} = \lambda_i \tag{A.140}$$

The *Rayleigh quotient* of two symmetric matrices $\mathbf{A}$ and $\mathbf{B}$ is a function of a vector $\mathbf{w}$ defined as:

$$R(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{A}\mathbf{w}}{\mathbf{w}^T\mathbf{B}\mathbf{w}} \tag{A.141}$$

To find the optimal $\mathbf{w}$ corresponding to the extremum (maximum or minimum) of $R(\mathbf{w})$, we find its derivative with respect to $\mathbf{w}$:

$$\frac{d}{d\mathbf{w}}R(\mathbf{w}) = \frac{2\mathbf{A}\mathbf{w}(\mathbf{w}^T\mathbf{B}\mathbf{w}) - 2\mathbf{B}\mathbf{w}(\mathbf{w}^T\mathbf{A}\mathbf{w})}{(\mathbf{w}^T\mathbf{B}\mathbf{w})^2} \tag{A.142}$$

Setting it to zero we get

$$\mathbf{A}\mathbf{w}(\mathbf{w}^T\mathbf{B}\mathbf{w}) = \mathbf{B}\mathbf{w}(\mathbf{w}^T\mathbf{A}\mathbf{w}), \quad \text{i.e.} \quad \mathbf{A}\mathbf{w} = \frac{\mathbf{w}^T\mathbf{A}\mathbf{w}}{\mathbf{w}^T\mathbf{B}\mathbf{w}}\mathbf{B}\mathbf{w} = R(\mathbf{w})\mathbf{B}\mathbf{w} = \lambda\mathbf{B}\mathbf{w} \tag{A.143}$$

The second equation is the generalized eigenequation of which $\lambda = R(\mathbf{w})$ is the eigenvalue and $\mathbf{w}$ the corresponding eigenvector. Solving this generalized eigenvalue problem we get the eigenvector $\mathbf{w}$ corresponding to the maximum or minimum eigenvalue, the Rayleigh quotient $\lambda = R(\mathbf{w})$.

A.3.4 Normal Matrices and Diagonalizability

Theorem: The product of two unitary matrices is unitary.

Proof: Let $\mathbf{U}$ and $\mathbf{V}$ be unitary, i.e., $\mathbf{U}^* = \mathbf{U}^{-1}$ and $\mathbf{V}^* = \mathbf{V}^{-1}$, then $\mathbf{UV}$ is unitary:

$$(\mathbf{UV})^* = \mathbf{V}^*\mathbf{U}^* = \mathbf{V}^{-1}\mathbf{U}^{-1} = (\mathbf{UV})^{-1} \tag{A.144}$$

Theorem: Two square matrices $\mathbf{A}$ and $\mathbf{B}$ are simultaneously diagonalizable if and only if they commute.

Proof

- Let $\mathbf{A}$ and $\mathbf{B}$ be simultaneously diagonalizable by $\mathbf{R}$

$$\mathbf{R}^{-1}\mathbf{AR} = \mathbf{\Lambda}_A, \qquad \mathbf{R}^{-1}\mathbf{BR} = \mathbf{\Lambda}_B \tag{A.145}$$

  then

$$\begin{aligned}\mathbf{AB} &= (\mathbf{R}\mathbf{\Lambda}_A\mathbf{R}^{-1})(\mathbf{R}\mathbf{\Lambda}_B\mathbf{R}^{-1}) = (\mathbf{R}\mathbf{\Lambda}_A\mathbf{\Lambda}_B\mathbf{R}^{-1}) \\ &= (\mathbf{R}\mathbf{\Lambda}_B\mathbf{\Lambda}_A\mathbf{R}^{-1}) = (\mathbf{R}\mathbf{\Lambda}_B\mathbf{R}^{-1})(\mathbf{R}\mathbf{\Lambda}_A\mathbf{R}^{-1}) \\ &= \mathbf{BA}\end{aligned} \tag{A.146}$$

- Let $\mathbf{A}$ and $\mathbf{B}$ commute, i.e., $\mathbf{AB} = \mathbf{BA}$. Assuming $\mathbf{u}$ is an eigenvector of $\mathbf{A}$ corresponding to eigenvalue λ, i.e., $\mathbf{Au} = \lambda\mathbf{u}$, then

$$\mathbf{ABu} = \mathbf{B}(\mathbf{Au}) = \lambda\mathbf{Bu} \tag{A.147}$$

  we see that $\mathbf{Bu}$ is also an eigenvector of $\mathbf{A}$ corresponding to the same eigenvalue λ, i.e., $\mathbf{Bu}$ must be a scaled version of $\mathbf{u}$ (in the same 1-D space): $\mathbf{Bu} = \gamma\mathbf{u}$, i.e., $\mathbf{u}$ is also an eigenvector of $\mathbf{B}$.

Theorem: A matrix is normal if and only if it is unitarily diagonalizable.

Proof

- If $\mathbf{A}$ is unitarily diagonalizable:

$$\mathbf{AU} = \mathbf{U\Lambda}, \qquad \mathbf{U}^{-1}\mathbf{AU} = \mathbf{U}^*\mathbf{AU} = \mathbf{\Lambda}, \qquad \mathbf{A} = \mathbf{U\Lambda U}^* \tag{A.148}$$

  where $\mathbf{U}^* = \mathbf{U}^{-1}$ is unitary and $\mathbf{\Lambda}$ is a diagonal matrix satisfying $\mathbf{\Lambda}^*\mathbf{\Lambda} = \mathbf{\Lambda\Lambda}^*$, then $\mathbf{A}$ is normal:

$$\begin{aligned}\mathbf{AA}^* &= (\mathbf{U\Lambda U}^*)(\mathbf{U\Lambda U}^*)^* = (\mathbf{U\Lambda U}^*)(\mathbf{U\Lambda}^*\mathbf{U}^*) = \mathbf{U\Lambda\Lambda}^*\mathbf{U}^* \\ &= \mathbf{U\Lambda}^*\mathbf{\Lambda U}^* = (\mathbf{U\Lambda}^*\mathbf{U}^*)(\mathbf{U\Lambda U}^*) = \mathbf{A}^*\mathbf{A}\end{aligned}$$

- If $\mathbf{A}$ is normal, then it is diagonalizable by a unitary matrix. First we show any matrix $\mathbf{A}$ can be written as

$$\mathbf{A} = \mathbf{B} + i\mathbf{C} \tag{A.149}$$

  where

$$\mathbf{B} = \frac{1}{2}(\mathbf{A} + \mathbf{A}^*) = \mathbf{B}^*, \qquad \mathbf{C} = -\frac{1}{2}(\mathbf{A} - \mathbf{A}^*) = \mathbf{C}^*, \tag{A.150}$$

are both Hermitian, and diagonalizable by a unitary matrix. As $\mathbf{A}$ is normal, we have

$$0 = \mathbf{A}\mathbf{A}^* - \mathbf{A}^*\mathbf{A} = (\mathbf{B}+i\mathbf{C})(\mathbf{B}-i\mathbf{C}) - (\mathbf{B}-i\mathbf{C})(\mathbf{B}+i\mathbf{C}) = 2i(\mathbf{C}\mathbf{B}-\mathbf{B}\mathbf{C}) \tag{A.151}$$

We see that $\mathbf{C}\mathbf{B} = \mathbf{B}\mathbf{C}$, i.e., $\mathbf{B}$ and $\mathbf{C}$ commute, and they can be simultaneously diagonalized by some unitary matrix $\mathbf{U}$:

$$\mathbf{U}^*\mathbf{B}\mathbf{U} = \mathbf{\Lambda}_B, \qquad \mathbf{U}^*\mathbf{C}\mathbf{U} = \mathbf{\Lambda}_C, \tag{A.152}$$

and so can $\mathbf{A} = \mathbf{B} + i\mathbf{C}$:

$$\mathbf{U}^*\mathbf{A}\mathbf{U} = \mathbf{U}^*(\mathbf{B}+i\mathbf{C})\mathbf{U} = \mathbf{\Lambda}_B + i\mathbf{\Lambda}_C = \mathbf{\Lambda}_A \tag{A.153}$$

Q.E.D.

If $\mathbf{A}$ is normal, i.e., it commutes with its conjugate transpose $\mathbf{A}^*\mathbf{A} = \mathbf{A}\mathbf{A}^*$, then $\mathbf{A}$ and $\mathbf{A}^*$ can simultaneously diagonalized by their unitary eigenvector matrix $\mathbf{U}$:

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^*, \qquad \mathbf{A}^* = \mathbf{U}\overline{\mathbf{\Lambda}}\mathbf{U}^* \tag{A.154}$$

A.3.5 Singular Value Decomposition

The eigenvalue decomposition of a square matrix $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$ is of great importance and widely used. However, for an $m \times n$ non-square matrix $\mathbf{A}$, no eigenvalues and eigenvector exist. In this case, we can still find its *singular values* and the corresponding left and right *singular vectors*, and then carry out *singular value decomposition (SVD)*.

Theorem: An $m \times n$ matrix $\mathbf{A}$ of rank $r \leq \min(m, n)$ can be diagonalized by two orthogonal matrices $\mathbf{U}_{m\times m}$ and $\mathbf{V}_{n\times n}$:

$$\mathbf{U}^T\mathbf{A}\mathbf{V} = \mathbf{\Sigma} \qquad \text{or} \qquad \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{A.155}$$

where

- $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_m]$ is an $m \times m$ matrix composed of the orthogonal eigenvectors of the symmetric matrix $\mathbf{A}\mathbf{A}^T$, also called the *left singular vectors* of $\mathbf{A}$:

$$\mathbf{U}^T(\mathbf{A}\mathbf{A}^T)\mathbf{U} = \mathbf{\Lambda} \qquad \text{or} \qquad \mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T \tag{A.156}$$

  where $\mathbf{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_n)$ is the e igenvalue matrix of $\mathbf{A}\mathbf{A}^T$.
- $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_n]$ is an $n \times n$ matrix composed of the orthogonal eigenvectors of the symmetric matrix $\mathbf{A}^T\mathbf{A}$, also called the *right singular vectors* of $\mathbf{A}$:

$$\mathbf{V}^T(\mathbf{A}^T\mathbf{A})\mathbf{V} = \mathbf{\Lambda} \qquad \text{or} \qquad \mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T \tag{A.157}$$

  Note that $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ have the same eigenvalue matrix $\mathbf{\Lambda}$ (see proof below).

- $\mathbf{\Sigma} = \mathbf{\Lambda}^{1/2}$ is an $m \times n$ diagonal matrix of r non-zero values $\sigma_i = \sqrt{\lambda_i}$ $(i = 1, \ldots, r)$, the square roots of the eigenvalues of $\mathbf{A}\mathbf{A}^T$ or $\mathbf{A}^T\mathbf{A}$, defined as the *singular values* of $\mathbf{A}$:

$$\mathbf{\Lambda}^{1/2} = \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \ddots & \\ & & \sqrt{\lambda_r} \end{bmatrix} = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_r \end{bmatrix} = \mathbf{\Sigma} \quad \text{(A.158)}$$

Proof:

Pre-multiplying $\mathbf{A}$ on both sides of the eigenequation of $\mathbf{A}^T\mathbf{A}$:

$$(\mathbf{A}^T\mathbf{A})\mathbf{V} = \mathbf{V}\mathbf{\Lambda} \quad \text{(A.159)}$$

we get

$$(\mathbf{A}\mathbf{A}^T)\mathbf{A}\mathbf{V} = \mathbf{A}\mathbf{V}\mathbf{\Lambda}, \quad \text{(A.160)}$$

Note that $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ have the same eigenvalue matrix $\mathbf{\Lambda}$, and $\mathbf{A}\mathbf{V}$ is the eigenvector matrix of $\mathbf{A}\mathbf{A}^T$. As the columns (or rows) of $\mathbf{A}\mathbf{V}$ are not normalized, $\mathbf{A}\mathbf{V}$ is not orthogonal:

$$(\mathbf{A}\mathbf{V})^T(\mathbf{A}\mathbf{V}) = \mathbf{V}^T\mathbf{A}^T\mathbf{A}\mathbf{V} = \mathbf{V}^T\mathbf{V}\mathbf{\Lambda} = \mathbf{\Lambda} \neq \mathbf{I} \quad \text{(A.161)}$$

If we post-multiply $\mathbf{A}\mathbf{V}$ by $\mathbf{\Lambda}^{-1/2}$, its ith column is scaled by $1/\sqrt{\lambda_i}$ $(i = 1, \cdots, n)$ and thereby normalized, the resulting matrix becomes orthogonal:

$$\left(\mathbf{A}\mathbf{V}\mathbf{\Lambda}^{-1/2}\right)^T \left(\mathbf{A}\mathbf{V}\mathbf{\Lambda}^{-1/2}\right) = \mathbf{\Lambda}^{-1/2}\mathbf{V}^T\mathbf{A}^T\mathbf{A}\mathbf{V}\mathbf{\Lambda}^{-1/2}$$
$$= \mathbf{\Lambda}^{-1/2}\mathbf{V}^T\mathbf{V}\mathbf{\Lambda}\mathbf{\Lambda}^{-1/2} = \mathbf{\Lambda}^{-1/2}\mathbf{\Lambda}\mathbf{\Lambda}^{-1/2} = \mathbf{I}$$

We denote this matrix composed of normalized orthogonal eigenvectors of $\mathbf{A}\mathbf{A}^T$ by

$$\mathbf{A}\mathbf{V}\mathbf{\Lambda}^{-1/2} = \mathbf{U} \quad \text{(A.162)}$$

Post-multiplying $(\mathbf{V}\mathbf{\Lambda}^{-1/2})^{-1} = \Lambda^{1/2}\mathbf{V}^{-1}$ on both sides, we get the SVD of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}^{1/2}\mathbf{V}^{-1} = \mathbf{U}\mathbf{\Lambda}^{1/2}\mathbf{V}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \quad \text{(A.163)}$$

Q.E.D.

The SVD is illustrated graphically below for the two cases of $m < n$ and $m > n$:

The component form of $\mathbf{A}\mathbf{V}\mathbf{\Lambda}^{-1/2} = \mathbf{U}$ above is

$$\mathbf{A}\mathbf{v}_i \frac{1}{\sqrt{\lambda_i}} = \mathbf{u}_i \quad (i = 1, \cdots, n) \quad \text{(A.164)}$$

relating the eigenvectors $\mathbf{v}_i$ of $\mathbf{A}^T\mathbf{A}$ to the eigenvectors $\mathbf{u}_i$ of $\mathbf{A}\mathbf{A}^T$.

We further consider some special cases:

- If $\mathbf{A}^T = \mathbf{A}^{-1}$ is unitary, i.e., $\mathbf{A}^T\mathbf{A} = \mathbf{A}\mathbf{A}^T = \mathbf{I}$, its eigenvalue matrix is $\mathbf{\Lambda} = \mathbf{I}$, and all singular values are unity $\sigma_i = 1$ for all $i = 1, \cdots, n$.

- If $\mathbf{A}^T = \mathbf{A}$ is symmetric, i.e., $\mathbf{A}\mathbf{A}^T = \mathbf{A}^T\mathbf{A} = \mathbf{A}^2$, and its singular value decomposition is $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, i.e.,

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T, \qquad \mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V} \tag{A.165}$$

  On the other hand, the eigenvalue decomposition of $\mathbf{A}^2 = \mathbf{A}^T\mathbf{A} = \mathbf{A}\mathbf{A}^T$ can be written as:

$$\mathbf{A}^2 = \mathbf{W}\mathbf{\Lambda}^2\mathbf{W} \tag{A.166}$$

  where $\mathbf{\Lambda}$ is the eigenvalue matrix of $\mathbf{A}$. We see that $\mathbf{\Lambda}^2 = \mathbf{\Sigma}^2$, i.e., the singular values are the absolute values of its eigenvalues $\sigma_i = \sqrt{\lambda_i^2} = |\lambda_i|$ $(i = 1, \cdots, n)$, and the left and right eigenvectors are the same as the eigenvectors $\mathbf{U} = \mathbf{V} = \mathbf{W}$.
- If $\mathbf{A}$ is normal satisfying $\mathbf{A}^T\mathbf{A} = \mathbf{A}\mathbf{A}^T$ (e.g., symmetric, Hermitian, unitary, etc.), both $\mathbf{A}$ and $\mathbf{A}^T$ can be simultaneously diagonalized by their unitary eigenvector matrix $\mathbf{U} = \mathbf{V}$, i.e.,

$$\mathbf{A}\mathbf{u}_i = \lambda_i\mathbf{u}_i, \qquad \mathbf{A}\mathbf{u}_i = \overline{\lambda_i}\mathbf{u}_i, \qquad (i = 1, \cdots, n) \tag{A.167}$$

  or in matrix form

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T, \qquad \mathbf{A}^T = \mathbf{U}\overline{\mathbf{\Lambda}}\mathbf{U}^T \tag{A.168}$$

  we have

$$\mathbf{A}\mathbf{A}^T = (\mathbf{U}\mathbf{\Lambda}\mathbf{U}^T)(\mathbf{U}\overline{\mathbf{\Lambda}}\mathbf{U}^T) = \mathbf{U}\mathbf{\Lambda}\overline{\mathbf{\Lambda}}\mathbf{U}^T = \mathbf{U}|\mathbf{\Lambda}|^2\mathbf{U}^T \tag{A.169}$$

  i.e., the singular values of any norm matrix $\mathbf{A}$ are simply the modulus of its eigenvalues $\sigma_i(\mathbf{A}) = |\lambda_i(\mathbf{A})|$ $(i = 1, \cdots, n)$.

The $m \times n$ matrix $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ can be considered as a linear transformation that converts a vector $\mathbf{x} \in \mathbb{C}^n$ to another vector $\mathbf{y} \in \mathbb{C}^m$ in three steps:

$$\mathbf{y} = \mathbf{A}\mathbf{x} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\,\mathbf{x} \tag{A.170}$$

1. Rotate vector $\mathbf{x}$ by the orthogonal matrix $\mathbf{V}^T$:

$$\mathbf{y}_1 = \mathbf{V}^T\,\mathbf{x}. \tag{A.171}$$

2. Scale each dimension Y_k of $\mathbf{y}_1$ by a factor of σ_k $(k = 1, \ldots, r)$:

$$\mathbf{y}_2 = \mathbf{\Sigma}\,\mathbf{y}_1 = \mathbf{\Sigma}\mathbf{V}^T\,\mathbf{x}. \tag{A.172}$$

3. Rotate vector $\mathbf{y}_2$ by the orthogonal matrix $\mathbf{U}$:

$$\mathbf{y} = \mathbf{U}\mathbf{y}_2 = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\,\mathbf{x} = \mathbf{A}\mathbf{x}. \tag{A.173}$$

The figure below illustrates the transformation of the three vertices of a triangle in 2-D space by a matrix $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$, which first rotates the vertices by 45 degrees CCW, scale horizontally and vertically by a factor of 3 and 2, respectively, and then rotate CW by 30 degrees.

The SVD has many applications, two of which are considered below.

The SVD equation

$$\mathbf{U}^T\mathbf{A}\mathbf{V} = \mathbf{\Lambda}^{1/2} = \mathbf{\Sigma} \tag{A.174}$$

can be considered as the forward SVD transform. Pre-multiplying $\mathbf{U}$ and post-multiplying $\mathbf{V}^T$ on both sides, we get the inverse transform:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = [\mathbf{u}_1, \cdots\cdots\cdots, \mathbf{u}_m] \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \vdots \\ \vdots \\ \mathbf{v}_n^T \end{bmatrix} = \sum_{i=1}^{r} \sigma_i [\mathbf{u}_i \mathbf{v}_i^T] \tag{A.175}$$

by which the original matrix $\mathbf{A}$ is represented as a linear combination of r matrices $[\mathbf{u}_i\mathbf{v}_i^T]$ weighted by the singular values $\sqrt{\lambda_i}$ $(i = 1, \ldots, r)$. We can rewrite both the forward and inverse SVD transform as a pair of forward and inverse transforms:

$$\begin{cases} \mathbf{\Sigma} = \mathbf{\Lambda}^{1/2} = \mathbf{U}^T\mathbf{A}\mathbf{V} \\ \mathbf{A} = \mathbf{U}\mathbf{\Lambda}^{1/2}\mathbf{V}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T. \end{cases} \tag{A.176}$$

This result indicate that the SVD can be used for data compression, if the 2-D data stored in an $m \times n$ matrix $\mathbf{A}$ can be approximated by keeping in the summation above only $k \ll r \leq min(m, n)$ terms corresponding to the k greatest singular values. The amount of data can be significantly reduced from $m \times n$ to no more than $k(m + n + 1)$.

Also, the pseudo-inverse of any matrix (non-square, not full-rank) can be found given its SVD

$$\mathbf{A} = \mathbf{U}\mathbf{\Lambda}^{1/2}\mathbf{V}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{A.177}$$

as

$$\mathbf{A}^- = (\mathbf{V}^T)^{-1}\mathbf{\Sigma}^-\mathbf{U}^{-1} = \mathbf{V}\mathbf{\Sigma}^-\mathbf{U}^T \tag{A.178}$$

where $\mathbf{\Sigma}^-$ is pseudo-inverse of $\mathbf{\Sigma}$, composed of the reciprocals $1/\sigma_k$ of the R singular values along the diagonal. The pseudo-inverse matrix is needed to find optimal solution of an inconsistent linear equation system.

A.4 Vector and Matrix Calculations

A.4.1 Vector Norms

The norm of a vector $\mathbf{x}$, denoted by $||\mathbf{x}||$, can be intuitively interpreted as its "size". For example, the norm of a real number $x \in \mathbb{R}$ in the 1-D real space is its absolute value $|x|$, or its distance to the origin, and the norm of a complex

number $z = x + jy \in \mathbb{C}$ is its modulus $|z| = \sqrt{x^2 + y^2}$, its Euclidean distance to the origin. Here is the most general definition of a vector norm:

Definition: The norm of a vector $\mathbf{x} \in V$ in vector space V is a real non-negative value representing the length or magnitude of the vector. Specifically, the norm of $\mathbf{x} \in V$ satisfy the following three conditions:

- Positivity:

$$||\mathbf{x}|| \geq 0, \qquad ||\mathbf{x}|| = 0 \quad \text{iff} \quad \mathbf{x} = \mathbf{0} \tag{A.179}$$

- Homogeneity:

$$||a\mathbf{x}|| = |a| \, ||\mathbf{x}||, \qquad a \in \mathbb{R} \quad \text{or} \quad \mathbb{C} \tag{A.180}$$

- Triangle inequality:

$$||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}|| \tag{A.181}$$

The triangle inequality can also be expressed in alternative forms. If we define $\mathbf{z} = -\mathbf{y}$, the triangle inequality becomes

$$||\mathbf{x} - \mathbf{z}|| \leq ||\mathbf{x}|| + ||\mathbf{z}|| \tag{A.182}$$

If we further define $\mathbf{u} = \mathbf{x} - \mathbf{z}$, i.e., $\mathbf{z} = \mathbf{x} - \mathbf{u}$, then the above becomes

$$||\mathbf{u}|| \leq ||\mathbf{x}|| + ||\mathbf{x} - \mathbf{u}||, \qquad \text{i.e.} \qquad ||\mathbf{u}|| - ||\mathbf{x}|| \leq ||\mathbf{x} - \mathbf{u}|| = ||\mathbf{u} - \mathbf{x}|| \tag{A.183}$$

Combining the two results above, we get

$$||\mathbf{x}|| - ||\mathbf{y}|| \leq ||\mathbf{x} - \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}|| \tag{A.184}$$

The first equality holds if $\mathbf{x}$ and $\mathbf{y}$ are in the same direction, the second equality holds if they are in opposite direction. More specially when $\mathbf{y} = \mathbf{0}$, both equalities hold.

The p-norms of an n-D vector $\mathbf{x} = [x_1, \cdots, x_n]^T$ and a function $f(x)$ are defined as:

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \qquad ||f(x)||_p = \left(\int |f(x)|^p \, dx \right)^{1/p} \tag{A.185}$$

The p-norm satisfies the three requirements in the definition of vector norm. The first two are trivially obvious, while the third one happens to be Minkowski's inequality (see appendix):

$$||\mathbf{x} + \mathbf{y}||_p \leq ||\mathbf{x}||_p + ||\mathbf{y}||_p \tag{A.186}$$

The p-norms corresponding to $p = 1$, $p = 2$, and $p = \infty$ are most commonly used:

- $p = 1$, the absolute sum of all n elements:

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i| \tag{A.187}$$

- $p = 2$, the Euclidean norm:

$$||\mathbf{x}||_2 = \left(\mathbf{x}^T\mathbf{x}\right)^{1/2} = \sqrt{\sum_{i=1}^{n} |x_i|^2} \tag{A.188}$$

- $p = \infty$, the maximum absolute value of all n elements:

$$||\mathbf{x}||_\infty = \left(\sum_{i=1}^{n} |x_i|^\infty\right)^{1/\infty} = \max\{|x_1|, \cdots, |x_n|\} \tag{A.189}$$

Out of the three p-norms, the Euclidean 2-norm is the only one that is *unitary invariant*, i.e., it is conserved or invariant under any unitary transform $\mathbf{y} = \mathbf{R}\mathbf{x}$, where $\mathbf{R}$ is a unitary matrix satisfying $\mathbf{R}^{-1} = \mathbf{R}^*$:

$$||\mathbf{y}||_2^2 = \mathbf{y}^*\mathbf{y} = (\mathbf{R}\mathbf{x})^*(\mathbf{R}\mathbf{x}) = \mathbf{x}^*\left(\mathbf{R}^*\mathbf{R}\right)\mathbf{x} = \mathbf{x}^*\mathbf{x} = ||\mathbf{x}||_2^2 \tag{A.190}$$

i.e., the length of the vector is not changed by any unitary transform (such as rotation, when $\mathbf{R}$ is a rotation matrix).

Definition: Two norms $||\mathbf{x}||_a$ and $||\mathbf{x}||_b$ are equivalent if there exist two positive real constants c and C so that

$$c\,||\mathbf{x}||_b \le ||\mathbf{x}||_a \le C\,||\mathbf{x}||_b \tag{A.191}$$

Note that this relationship can also be written as

$$\frac{1}{C}\,||\mathbf{x}||_a \le ||\mathbf{x}||_b \le \frac{1}{c}\,||\mathbf{x}||_a \tag{A.192}$$

Theorem: All different norms $\mathbf{x}$ are equivalent.

Proof

- We first show that equivalence is transitive, i.e., if both $||\mathbf{x}||_a$ and $||\mathbf{x}||_b$ are equivalent to $||\mathbf{x}||_1$:

$$c\,||\mathbf{x}||_1 \le ||\mathbf{x}||_a \le C\,||\mathbf{x}||_1, \quad \text{and} \quad d\,||\mathbf{x}||_1 \le ||\mathbf{x}||_b \le D\,||\mathbf{x}||_1, \tag{A.193}$$

  then they are equivalent to each other.

  From the first equation we get $||\mathbf{x}||_1 \le ||\mathbf{x}||_a/c$ and $||\mathbf{x}||_1 \ge ||\mathbf{x}||_a/C$. Substituting these into the right and left hand sides of the second equation respectively yields:

$$\frac{d}{C}\,||\mathbf{x}||_a \le ||\mathbf{x}||_b \le \frac{D}{c}\,||\mathbf{x}||_a, \tag{A.194}$$

  i.e., $||\mathbf{x}||_a$ and $||\mathbf{x}||_b$ are equivalent.
- If we can further show that an arbitrary norm $||\mathbf{x}||$ is equivalent to $||\mathbf{x}||_1$, i.e.,

$$c\,||\mathbf{x}||_1 \le ||\mathbf{x}|| \le C\,||\mathbf{x}||_1 \tag{A.195}$$

  then according to the transitivity property shown above, all norms are equivalent.

  If $\mathbf{x} = \mathbf{0}$, it is obvious that the equalities above hold $||\mathbf{x}|| = ||\mathbf{x}||_1 = 0$. If $\mathbf{x} \neq \mathbf{0}$, $||\mathbf{x}||_1 > 0$, we can define a normalized vector $\mathbf{u} = \mathbf{x}/||\mathbf{x}||_1$ with

$||\mathbf{u}||_1 = 1$, so that the relationship for the equivalence above can be written in terms of $\mathbf{u}$ as

$$c = c||\mathbf{u}||_1 \leq ||\mathbf{u}|| \leq C||\mathbf{u}||_1 = C \tag{A.196}$$

i.e., to prove that all norms are equivalent all we need to show is that $||\mathbf{u}||$ is bounded both from above and below, as we will do below.

- We now prove that an arbitrary norm $f(\mathbf{x}) = ||\mathbf{x}||$ is a continuous function over $||\mathbf{x}||_1$, i.e., for any $\epsilon > 0$, there exists a $\delta > 0$ so that

$$\text{if} \quad ||\mathbf{x} - \mathbf{x}'||_1 < \delta, \quad \text{then} \quad ||\mathbf{x}|| - ||\mathbf{x}'|| \leq \epsilon \tag{A.197}$$

Both $\mathbf{x}$ and $\mathbf{x}'$ can be expressed in terms of a basis $\{\mathbf{b}_1, \cdots, \mathbf{b}_n\}$ that spans the space:

$$\mathbf{x} = \sum_{i=1}^{n} a_i \mathbf{b}_i, \qquad \mathbf{x}' = \sum_{i=1}^{n} a_i' \mathbf{b}_i \tag{A.198}$$

then we have

$$||\mathbf{x}-\mathbf{x}'|| = ||\sum_{i=1}^{n}(a_i - a_i')\mathbf{b}_i|| \leq \sum_{i=1}^{n}|a_i - a_i'|\,||\mathbf{b}_i|| \leq \sum_{i=1}^{n}|a_i - a_i'|\,\max_i ||\mathbf{b}_i|| = ||\mathbf{x}-\mathbf{x}'||_1\,\max_i ||\mathbf{b}_i|| \tag{A.199}$$

Now consider the alternative form of the triangle inequality of $||\mathbf{x}||$:

$$||\mathbf{x}|| - ||\mathbf{x}'|| \leq ||\mathbf{x} - \mathbf{x}'|| \leq ||\mathbf{x} - \mathbf{x}'||_1\,\max_i ||\mathbf{b}_i|| \tag{A.200}$$

If we let $\max_i ||\mathbf{b}_i|| = \epsilon/\delta$, the above becomes

$$||\mathbf{x}|| - ||\mathbf{x}'|| \leq ||\mathbf{x} - \mathbf{x}'||_1\,\frac{\epsilon}{\delta} \tag{A.201}$$

indicating that for any given ϵ, we can choose $\delta = \epsilon/\max_i ||\mathbf{b}_i||$ so that if $||\mathbf{x}-\mathbf{x}'||_1 \leq \delta$, then $||\mathbf{x}|| - ||\mathbf{x}'|| \leq \epsilon$, i.e., the norm $||\mathbf{x}||$ is indeed continuous over $||\mathbf{x}||_1$.

- Finally, according to the *extreme value theorem*, a continuous function, such as $f(\mathbf{x}) = ||\mathbf{x}||$, defined over a compact (closed and bounded) set, such as the unit sphere $||\mathbf{u}||_1 = 1$ in the n-D space, must have its maximum and minimum values:

$$c = \min_{||\mathbf{u}||_1 = 1} ||\mathbf{u}||, \quad C = \max_{||\mathbf{u}||_1 = 1} ||\mathbf{u}|| \tag{A.202}$$

i.e., $c \leq ||\mathbf{u}|| \leq C$, which is what we need to prove.

Q.E.D.

The three p-norms are equivalent:

$$\begin{aligned} ||\mathbf{x}||_\infty &\leq ||\mathbf{x}||_1 \leq n||\mathbf{x}||_\infty \\ ||\mathbf{x}||_\infty &\leq ||\mathbf{x}||_2 \leq \sqrt{n}||\mathbf{x}||_\infty \\ ||\mathbf{x}||_2 &\leq ||\mathbf{x}||_1 \leq \sqrt{n}||\mathbf{x}||_2 \end{aligned}$$

The distance between two vectors $\mathbf{x}, \mathbf{y} \in V$ is fined as the norm of their

difference $d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||$. In particular, when $\mathbf{y} = \mathbf{0}$, the distance $||\mathbf{x} - \mathbf{0}|| = ||\mathbf{x}||$ between $\mathbf{x}$ and the origin $\mathbf{0}$ of the space is the norm of $\mathbf{x}$. Specifically, $d(\mathbf{x}, \mathbf{y})$ can be defined based on the p-norm:

$$d_p(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_p = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}, \quad (1 \leq p \leq \infty) \tag{A.203}$$

Specially, when

- $p = 1$, $d_1(\mathbf{x}, \mathbf{y})$ is the city block (Manhattan) distance:

$$d_1(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_1 = \sum_{i=1}^{n} |x_i - y_i| \tag{A.204}$$

- $p = 2$, $d_2(\mathbf{x}, \mathbf{y})$ is the Euclidean distance:

$$d_2(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2} \tag{A.205}$$

- $p = \infty$, $d_\infty(\mathbf{x}, \mathbf{y})$ is the Chebyshev distance:

$$d_\infty(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_\infty = \max_i |x_i - y_i| \tag{A.206}$$

The three unit "circles" or "spheres", are formed by all points $\mathbf{x}$ of unity norm $||\mathbf{x}||_p = 1$ with unity distance to the origin (blue, black, and red for d_∞, d_2, and d_1, respectively).

A.4.2 Matrix Norms

The norm of an $m \times n$ matrix $\mathbf{A}$ is any function that maps $\mathbf{A}$ to a real number $||\mathbf{A}||$ that satisfies the following required properties:

- Positivity:

$$||\mathbf{A}|| \geq 0, \qquad ||\mathbf{A}|| = 0 \quad \text{iff} \quad \mathbf{A} = \mathbf{0} \tag{A.207}$$

- Homogeneity:

$$||a\mathbf{A}|| = |a| \, ||\mathbf{A}|| \tag{A.208}$$

- Triangle inequality:

$$||\mathbf{A} + \mathbf{B}|| \leq ||\mathbf{A}|| + ||\mathbf{B}||, \qquad \text{or} \qquad ||\mathbf{A}|| - ||\mathbf{B}|| \leq ||\mathbf{A} - \mathbf{B}|| \tag{A.209}$$

  The second expression can be obtained by replacing $\mathbf{A}$ by $\mathbf{A} - \mathbf{B}$ in the first expression.

In addition to the three required properties for matrix norms, some of them also satisfy these additional properties (not necessarily satisfied by other matrix norms):

- Subordinance:

$$||\mathbf{Ax}|| \leq ||\mathbf{A}|| \cdot ||\mathbf{x}|| \tag{A.210}$$

- submultiplicativity:

$$||\mathbf{AB}|| \leq ||\mathbf{A}|| \cdot ||\mathbf{B}|| \tag{A.211}$$

We now consider some commonly used matrix norms:

- **The element-wise norms**

If we treat the $m \times n$ elements of $\mathbf{A}$ are the elements of an mn-dimensional vector, then the p-norm of this vector can be used as the p-norm of $\mathbf{A}$:

$$||\mathbf{A}||_p = \left\{ \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^p \right\}^{1/p} \tag{A.212}$$

Specially, we consider the following three cases for $p = 1,\ 2,\ \infty$.

– $||\mathbf{A}||_1$ is the absolute sum of all elements of $\mathbf{A}$:

$$||\mathbf{A}||_1 = \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}| \tag{A.213}$$

– $||\mathbf{A}||_\infty$ is the *maximum norm*, the maximum absolute value among all mn elements of $\mathbf{A}$:

$$||\mathbf{A}||_\infty = \max\{|a_{ij}|,\ \ 1 \leq i \leq m,\ 1 \leq j \leq n\} \tag{A.214}$$

– $||\mathbf{A}||_2$ is the *Frobenius norm*

$$||\mathbf{A}||_2 = ||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2} = \sqrt{tr(\mathbf{A}^*\mathbf{A})} = \sqrt{\sum_{i=1}^{R} \lambda_i} = \sqrt{\sum_{i=1}^{R} \sigma_i^2} \tag{A.215}$$

where $r \leq \min(m, n)$ is the rank of $\mathbf{A}$, $\lambda_i = \sigma_2^2$ is the ith non-zero eigenvalues of the positive, semi-definite matrix $\mathbf{A}^*\mathbf{A}$, and $\sigma_i = \sqrt{\lambda_i}$ the ith singular value of $\mathbf{A}$ $(i = 1, \cdots, r)$. (This Frobenius norm $||\mathbf{A}||_2$ is implemented in Matlab by the function `norm(A,'fro')`.)

The Frobenius norm of a unitary (orthogonal if real) matrix $\mathbf{R}$ satisfying $\mathbf{R}^* = \mathbf{R}^{-1}$ or $\mathbf{R}^*\mathbf{R} = \mathbf{R}^{-1}\mathbf{R} = \mathbf{I}$ is:

$$||\mathbf{R}||_F^2 = tr(\mathbf{R}^*\mathbf{R}) = tr\mathbf{I} = n \tag{A.216}$$

The Frobenius norm is the only one out of the above three matrix norms that is *unitary invariant*, i.e., it is conserved or invariant under a unitary transformation (such as a rotation) $\mathbf{B} = \mathbf{R}^*\mathbf{AR}$:

$$\begin{aligned} ||\mathbf{B}||_F^2 &= \mathrm{tr}\,(\mathbf{B}^*\mathbf{B}) = \mathrm{tr}\left[(\mathbf{R}^*\mathbf{AR})^*\,(\mathbf{R}^*\mathbf{AR})\right] = \mathrm{tr}\,(\mathbf{R}^*\mathbf{A}^*\mathbf{RR}^*\mathbf{AR}) \\ &= \mathrm{tr}\,(\mathbf{R}^*\mathbf{A}^*\mathbf{AR}) = \mathrm{tr}\,(\mathbf{A}^*\mathbf{A}) = ||\mathbf{A}||_F^2 = ||\mathbf{A}||_F^2 \end{aligned}$$

where we have used the property of the trace $\mathrm{tr}\,(\mathbf{AB}) = \mathrm{tr}\,(\mathbf{BA})$.

- **The natural matrix norms**

  The natural norm $||\mathbf{A}||$ of matrix $\mathbf{A}$ induced by (subordinate to) the vector norm $||\mathbf{x}||$ is defined as

$$||\mathbf{A}|| = \sup_{||\mathbf{x}||=1} \{||\mathbf{A}\mathbf{x}||\} = \sup_{\mathbf{x}\neq\mathbf{0}} \{||\mathbf{A}\mathbf{x}||/||\mathbf{x}||\} \tag{A.217}$$

  Specially the norm of the identity matrix $\mathbf{A} = \mathbf{I}$ is

$$||\mathbf{I}|| = \sup_{||\mathbf{x}||=1} \{||\mathbf{I}\mathbf{x}||\} = \sup_{||\mathbf{x}||=1} \{||\mathbf{x}||\} = 1 \tag{A.218}$$

  We now prove that the induced matrix norm satisfies all five properties given above. (Recall $|a+b| \leq |a| + |b|$, $|ab| \leq |a|\,|b|$.)

  1.Positivity: $||\mathbf{A}|| > 0$ if $\mathbf{A} \neq \mathbf{0}$, this is trivially obvious.

  2.Homogeneity: $||a\mathbf{A}|| = |a|\,||\mathbf{A}||$

$$||a\mathbf{A}|| = \sup_{||\mathbf{x}||=1} \{||a\mathbf{A}\mathbf{x}||\} = \sup_{||\mathbf{x}||=1} \{|a|\,||\mathbf{A}\mathbf{x}||\} = |a| \sup_{||\mathbf{x}||=1} \{||\mathbf{A}\mathbf{x}||\} = |a|\,||\mathbf{A}|| \tag{A.219}$$

  3.Triangle inequality: $||\mathbf{A}+\mathbf{B}|| \leq ||\mathbf{A}|| + ||\mathbf{B}||$

$$\begin{aligned} ||\mathbf{A}+\mathbf{B}|| &= \sup_{||\mathbf{x}||=1} \{||(\mathbf{A}+\mathbf{B})\mathbf{x}||\} = \sup_{||\mathbf{x}||=1} \{||\mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{x}||\} \leq \sup_{||\mathbf{x}||=1} \{||\mathbf{A}\mathbf{x}|| + ||\mathbf{B}\mathbf{x}||\} \\ &\leq \sup_{||\mathbf{x}||=1} \{||\mathbf{A}\mathbf{x}||\} + \sup_{||\mathbf{x}||=1} \{||\mathbf{B}\mathbf{x}||\} = ||\mathbf{A}|| + ||\mathbf{B}|| \end{aligned}$$

  4.Subordinance: $||\mathbf{A}\mathbf{x}|| \leq ||\mathbf{A}||\,||\mathbf{x}||$

$$||\mathbf{A}||\,||\mathbf{x}|| = ||\mathbf{x}|| \sup_{\mathbf{y}} \{||\mathbf{A}\mathbf{y}||/||\mathbf{y}||\} \geq ||\mathbf{x}||\,||\mathbf{A}\mathbf{y}||/||\mathbf{y}|| \tag{A.220}$$

  As $\mathbf{y}$ is arbitrary, we let $\mathbf{y} = \mathbf{x}$ and get

$$||\mathbf{A}||\,||\mathbf{x}|| \geq ||\mathbf{A}\mathbf{x}|| \tag{A.221}$$

  In particular, if vector $\mathbf{x}$ is the eigenvector corresponding to the greatest eigenvalue λ_{max} of $\mathbf{A}^*\mathbf{A}$:

$$(\mathbf{A}^*\mathbf{A})\,\mathbf{x} = \lambda_{max}\,\mathbf{x} = ||\mathbf{A}||^2\,\mathbf{x} \tag{A.222}$$

  then the equality of the subordinance property holds. To see this, consider

$$||\mathbf{A}\mathbf{x}||^2 = \langle (\mathbf{A}\mathbf{x})^*,\, (\mathbf{A}\mathbf{x})\rangle = \mathbf{x}^*(\mathbf{A}^*\mathbf{A})\mathbf{x} = \mathbf{x}^*||\mathbf{A}||^2\mathbf{x} = ||\mathbf{A}||^2||\mathbf{x}||^2 \tag{A.223}$$

  taking square root on both sides we get $||\mathbf{A}\mathbf{x}|| = ||\mathbf{A}||\,||\mathbf{x}||$.

  5.Submultiplicativity: $||\mathbf{A}\mathbf{B}|| \leq ||\mathbf{A}||\,||\mathbf{B}||$

$$||\mathbf{A}\mathbf{B}|| = \sup_{||\mathbf{x}||=1} \{||\mathbf{A}\mathbf{B}\mathbf{x}||\} \leq \sup_{||\mathbf{x}||=1} \{||\mathbf{A}||\,||\mathbf{B}\mathbf{x}||\} = ||\mathbf{A}|| \sup_{||\mathbf{x}||=1} \{||\mathbf{B}\mathbf{x}||\} = ||\mathbf{A}||\,||\mathbf{B}|| \tag{A.224}$$

  In particular, if $\mathbf{A} = c\mathbf{B}^*$ (i.e., $\mathbf{A}$ and $\mathbf{B}$ are linearly dependent), then the equality of the submultiplicativity property holds. To see this, consider

$$||\mathbf{B}|| = \sqrt{\lambda_{max}(\mathbf{B}^*\mathbf{B})}, \quad ||\mathbf{A}|| = |c|\sqrt{\lambda_{max}(\mathbf{B}\mathbf{B}^*)} = |c|\sqrt{\lambda_{max}(\mathbf{B}^*\mathbf{B})} \tag{A.225}$$

and

$$||\mathbf{AB}|| = |c|\,||\mathbf{B}^*\mathbf{B}|| = |c|\,\sqrt{\lambda_{max}((\mathbf{B}^*\mathbf{B})(\mathbf{B}^*\mathbf{B}))} = |c|\,\sqrt{\lambda_{max}(\mathbf{B}^*\mathbf{B})^2} = |c|\,\lambda_{max}(\mathbf{B}^*\mathbf{B}) \tag{A.226}$$

i.e., $||\mathbf{AB}|| = ||\mathbf{A}||\,||\mathbf{B}||$.

Consider in particular the natural matrix norms $||\mathbf{A}||_p$ corresponding to the three vector p-norms $||\mathbf{x}||_p$ with $p = 1, 2, \infty$:

– When $p = 1$, $||\mathbf{A}||_1$ (`norm(A,1)` in Matlab) is the maximum absolute column sum:

$$||\mathbf{A}||_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \tag{A.227}$$

Proof: The 1-norm of vector $\mathbf{x}$ is $||\mathbf{x}||_1 = \sum_{j=1}^{n} |x_j|$, we have

$$||\mathbf{Ax}||_1 = \sum_{i=1}^{m} \left| \sum_{j=1}^{n} a_{ij}x_j \right| \le \sum_{i=1}^{m}\sum_{j=1}^{n} (|a_{ij}|\,|x_j|) = \sum_{j=1}^{n} \left(|x_j| \sum_{i=1}^{m} |a_{ij}| \right)$$
$$\le \left(\max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \right) \left(\sum_{j=1}^{n} |x_j| \right) = ||\mathbf{A}||_1\,||\mathbf{x}||_1$$

Assuming the kth column of $\mathbf{A}$ has the maximum absolute sum and $\mathbf{x}$ is normalized (as required in the definition) with $||\mathbf{x}|| = 1$, we have

$$||\mathbf{A}||_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| = \sum_{i=1}^{m} |a_{ik}| \tag{A.228}$$

and

$$||\mathbf{Ax}||_1 \le ||\mathbf{A}||_1\,||\mathbf{x}||_1 = \sum_{i=1}^{m} |a_{ik}| \tag{A.229}$$

Now we show that the equality of the above can be achieved, i.e., $||\mathbf{Ax}||_1$ is maximized, if we choose $\mathbf{x} = \mathbf{e}_k = [0, \cdots, 1, \cdots, 0]^T$, the kth unit vector (normalized):

$$||\mathbf{Ax}||_1 = \sum_{i=1}^{m} \left| \sum_{j=1}^{n} a_{ij}x_j \right| = \sum_{i=1}^{m} |a_{ik}| = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \tag{A.230}$$

i.e., $\mathbf{x} = \mathbf{e}_k$ is the vector among all other normalized vectors that maximizes $||\mathbf{Ax}||$ as required in the definition, and the resulting maximum $||\mathbf{A}||_1$ is indeed $\max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|$. We therefore have

$$||\mathbf{A}||_1 = \sup_{||\mathbf{x}||_1=1} \{||\mathbf{Ax}||_1\} = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}| \tag{A.231}$$

– When $p = \infty$, $||\mathbf{A}||_\infty$ (`norm(A,inf)` in Matlab) is the maximum absolute row sum:

$$||\mathbf{A}||_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}| \tag{A.232}$$

Proof: When $p = \infty$, $\mathbf{x}$ is normalized if $||\mathbf{x}||_\infty = \max\{x_1, \cdots, x_n\} = 1$. The norm of vector $\mathbf{Ax}$ is:

$$||\mathbf{Ax}||_\infty = \max\left\{\sum_{j=1}^{n}|a_{ij}x_j|, \cdots, \sum_{j=1}^{n}|a_{mj}x_j|\right\} \tag{A.233}$$

which can be maximized by any normalized vector with $||\mathbf{x}||_\infty = \max\{x_1, \cdots, x_n\} = 1$ to become

$$||\mathbf{Ax}||_\infty = \max\left\{\sum_{j=1}^{n}|a_{ij}|, \cdots, \sum_{j=1}^{n}|a_{mj}|\right\} \tag{A.234}$$

We therefore have

$$||\mathbf{A}||_\infty = \sup_{||\mathbf{x}||_\infty=1}\{||\mathbf{Ax}||_\infty\} = \max_i \sum_{j=1}^{n}|a_{ij}| \tag{A.235}$$

For example, for any stochastic matrix $\mathbf{P}$ satisfying $\sum_j P_{ij} = 1$, $||\mathbf{P}||_\infty = 1$.

– When $p = 2$, $||\mathbf{A}||_2$ (`norm(A,2)` or `norm(A)` in Matlab), also called the *spectral norm*, is the greatest singular value of $\mathbf{A}$, square root of the greatest eigenvalue of $\mathbf{A}^*\mathbf{A}$, i.e., its spectral radius $\rho(\mathbf{A}^*\mathbf{A}) = \lambda_{max}(\mathbf{A}^*\mathbf{A})$:

$$||\mathbf{A}||_2 = \sigma_{max}(\mathbf{A}) = \sqrt{\lambda_{max}(\mathbf{A}^*\mathbf{A})} = \sqrt{\rho(\mathbf{A}^*\mathbf{A})} \tag{A.236}$$

where $\lambda_{max}(\mathbf{A}^*\mathbf{A}) = \sigma^2_{max}(\mathbf{A})$ is the maximal eigenvalue of $\mathbf{A}^*\mathbf{A}$.

Proof:

First we consider the eigenequation of $\mathbf{A}^*\mathbf{A}$:

$$\mathbf{A}^*\mathbf{A}\mathbf{v} = \lambda_i\mathbf{v}_i, \quad (i = 1, \cdots, n) \quad \text{or} \quad \mathbf{A}^*\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Lambda}, \quad \text{or} \quad \mathbf{A}^*\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^* \tag{A.237}$$

where

$$\mathbf{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_n), \quad \text{and} \quad \mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_n] \tag{A.238}$$

are the eigenvector and eigenvalue matrices of $\mathbf{A}^*\mathbf{A}$. As $\mathbf{A}^*\mathbf{A}$ is symmetric and positive definite, all of its eigenvalues are real and positive, and assumed to be sorted

$$\lambda_1 = \lambda_{max} \geq \lambda_2 \geq \cdots \geq \lambda_n = \lambda_{min} \geq 0 \tag{A.239}$$

and all corresponding eigenvectors are orthogonal and assumed to be normalized, i.e., $\mathbf{v}_i^*\mathbf{v}_j = \delta_{ij}$, or $\mathbf{V}^* = \mathbf{V}^{-1}$ is a unitary (orthogonal if real) matrix.

Next, consider the $p = 2$ norm of $\mathbf{Ax}$:

$$||\mathbf{Ax}||_2 = [(\mathbf{Ax})^*(\mathbf{Ax})]^{1/2} = (\mathbf{x}^*\mathbf{A}^*\mathbf{Ax})^{1/2} = (\mathbf{x}^*\mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^*\mathbf{x})^{1/2}$$
$$= (\mathbf{y}^*\boldsymbol{\Lambda}\mathbf{y})^{1/2} = \sqrt{\sum_{i=1}^{n} y_i^2\lambda_i}$$

Here $\mathbf{y} = \mathbf{V}^*\mathbf{x}$ is a unitary transform of $\mathbf{x}$ with the 2-norm $||\mathbf{y}||_2 = ||\mathbf{x}||_2$ conserved.
The right-hand side of the equation above is a weighted average of the n eigenvalues $\lambda_1, \cdots, \lambda_n$, which is maximized if they are weighted by a normalized vector $\mathbf{y} = [1, 0, \cdots, 0]^T$ with $||\mathbf{y}||_2 = 1$, by which the greatest eigenvalue λ_{max} is maximally weighted while all others are weighted by 0:

$$||\mathbf{Ax}||_2 \leq \sqrt{\lambda_{max}} \tag{A.240}$$

As $||\mathbf{x}||_2 = ||\mathbf{y}||_2 = 1$, we have

$$||\mathbf{A}||_2 = \sup_{||\mathbf{x}||_2=1} \{||\mathbf{Ax}||_2\} = \sqrt{\lambda_{max}} \tag{A.241}$$

Out of the three matrix norms defined above, the spectral norm is the only one that is *unitary invariant*, i.e., it is conserved or invariant under any unitary transform $\mathbf{B} = \mathbf{R}^*\mathbf{AR}$:

$$||\mathbf{B}||_2^2 = \lambda_{max}(\mathbf{B}^*\mathbf{B}) = \lambda_{max}\left[(\mathbf{R}^*\mathbf{AR})^*(\mathbf{R}^*\mathbf{AR})\right] = \lambda_{max}(\mathbf{R}^*\mathbf{A}^*\mathbf{RR}^*\mathbf{AR})$$
$$= \lambda_{max}(\mathbf{R}^*\mathbf{A}^*\mathbf{AR}) = \lambda_{max}(\mathbf{A}^*\mathbf{A}) = ||\mathbf{A}||_2^2$$

Here we have used the fact that the eigenvalues and eigenvectors are invariant under the unitary transform.

Example A.3

$$\mathbf{A} = \begin{bmatrix} 3 & -6 & 2 \\ 2 & 5 & 1 \\ -3 & 2 & 2 \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 3 & 2 & 1 \\ 2 & -3 & 0 \\ 1 & 0 & -1 \end{bmatrix} \tag{A.242}$$

$$||\mathbf{A}||_1 = \max\{|3| + |2| + |-3|,\ |-6| + |5| + |2|,\ |2| + |1| + |2|\} = \max\{8,\ 13,\ 5\} = 13$$
$$||\mathbf{A}||_\infty = \max\{|3| + |-6| + |2|,\ |2| + |5| + |1|,\ |-3| + |2| + |2|\} = \max\{11,\ 8,\ 7\} = 11$$

The eigenvalues of $\mathbf{A}^*\mathbf{A}$ are

$$\lambda_1 = 69.353, \quad \lambda_2 = 17.967, \quad \lambda_3 = 8.680 \tag{A.243}$$

The singular values of $\mathbf{A}$ are

$$\sigma_1 = \sqrt{\lambda_1} = 8.328, \quad \sigma_2 = \sqrt{\lambda_2} = 4.239, \quad \sigma_3 = \sqrt{\lambda_3} = 2.946 \tag{A.244}$$

The spectral norm of $\mathbf{A}$ is

$$||\mathbf{A}||_2 = \sigma_{max} = \sqrt{\lambda_{max}} = 8.328 \tag{A.245}$$

The eigenvector corresponding to greatest eigenvalue λ_1 is $\mathbf{x} = [0.285,\ -0.957,\ 0.057]^T$, which satisfies the equality $||\mathbf{Ax}|| = ||\mathbf{A}||\ ||\mathbf{x}||$.

- **The *Schatten norms***

  The Shatten norm is defined based on the singular values σ_i of $\mathbf{A}$ or the eigenvalues $\lambda_i = \sigma^2$ of $\mathbf{A}^*\mathbf{A}$:

$$||\mathbf{A}||_p = \left(\sum_{i=1}^{R} \sigma_i^p\right)^{1/p} = \left(\sum_{i=1}^{R} \left(\sqrt{\lambda_i}\right)^p\right)^{1/p} \tag{A.246}$$

  In particular, consider three common p values:

  – $p = 1$ is the *nuclear or trace* norm:

$$||\mathbf{A}||_1 = \sum_{i=1}^{R} \sigma_i = \sum_{i=1}^{R} \sqrt{\lambda_i} = \text{tr}\left(\sqrt{\mathbf{A}^*\mathbf{A}}\right) \tag{A.247}$$

  – $p = 2$ same as the Frobenius norm:

$$||\mathbf{A}||_2 = \sqrt{\sum_{i=1}^{R} \sigma_i^2} = \sqrt{\sum_{i=1}^{R} \lambda_i} = ||\mathbf{A}||_F \tag{A.248}$$

  – $p = \infty$ same as the spectral norm (the induced 2-norm), the spectral radius of $\mathbf{A}^*\mathbf{A}$.

$$||\mathbf{A}||_\infty = \max\{\sigma_1, \cdots, \sigma_n\} = \sigma_{max} = \sqrt{\lambda_{max}} \tag{A.249}$$

  As the eigenvalues and eigenvectors of $\mathbf{A}^*\mathbf{A}$ are invariant under unitary transform, the Schatten norms are unitary invariant as well.

All matrix norms defined above are equivalent according to the theorem previously discussed.

- The Frobenius norm $||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{R} \lambda_i}$ and the induced 2-norm $||\mathbf{A}||_2 = \sqrt{\lambda_{max}}$ are equivalent:

$$||\mathbf{A}||_2 \leq ||\mathbf{A}||_F \leq \sqrt{R}\ ||\mathbf{A}||_2 \tag{A.250}$$

  The equality on the left holds when all eigenvalues λ_i but one are zero, and the equality on the right holds when all λ_i are the same.

- The Frobenius norm $||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{R} \lambda_i}$ and the Schatten 1-norm $||\mathbf{A}||_1 = \sum_{i=1}^{R} \sqrt{\lambda_i}$ are equivalent:

$$||\mathbf{A}||_F \leq ||\mathbf{A}||_1 \leq \sqrt{R}\ ||\mathbf{A}||_F \tag{A.251}$$

  The equality on the left holds when all eigenvalues λ_i but one are zero, and the equality on the right holds when all λ_i are the same.

- The element-wise maximum norm $||\mathbf{A}||_{max} = \max_{i,j}\{|a_{ij}|\}$ and the Frobenius norm $||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$ are equivalent:

$$||\mathbf{A}||_{max} \leq ||\mathbf{A}||_F \leq \sqrt{mn}\,||\mathbf{A}||_{max} \tag{A.252}$$

  The equality on the left holds when all elements λ_i but one are zero, and the equality on the right holds when all elements are the same.

- $$\frac{1}{\sqrt{n}}||\mathbf{A}||_\infty \leq ||\mathbf{A}||_2 \leq \sqrt{m}||\mathbf{A}||_\infty \tag{A.253}$$

  Proof: Define an n-D vector $\mathbf{e} = [1, \cdots, 1]^T$, then the greatest absolute row sum of $\mathbf{A}$ is

$$||\mathbf{A}||_\infty = \max_i \sum_{j=1}^{n}|a_{ij}| = ||\mathbf{Ae}||_\infty \leq ||\mathbf{Ae}||_2 \leq ||\mathbf{A}||_2\,||\mathbf{e}||_2 = \sqrt{n}||\mathbf{A}||_2 \tag{A.254}$$

  i.e.,

$$\frac{1}{\sqrt{n}}||\mathbf{A}||_\infty \leq ||\mathbf{A}||_2 \tag{A.255}$$

- $$\frac{1}{\sqrt{m}}||\mathbf{A}||_1 \leq ||\mathbf{A}||_2 \leq \sqrt{n}||\mathbf{A}||_1 \tag{A.256}$$

Theorem: The spectral radius of a matrix is bounded by its matrix norm: $\rho(\mathbf{A}) \leq ||\mathbf{A}||$

Proof: Let λ and $\mathbf{u}$ be an eigenvalue and the corresponding normalized eigenvector of a square matrix $\mathbf{A}$, i.e., $\mathbf{Au} = \lambda\mathbf{u}$ and $||\mathbf{u}|| = 1$. As the induced norm, $||\mathbf{A}||$ satisfies

$$||\mathbf{A}|| = \sup_{||\mathbf{x}||=1}\{||\mathbf{Ax}|\} \geq ||\mathbf{Au}|| = ||\lambda\mathbf{u}|| = |\lambda|\,||\mathbf{u}|| = |\lambda| \tag{A.257}$$

But λ is an arbitrary eigenvalue, it can be the one with the maximum absolute value $|\lambda|$, i.e., the spectral radius $\rho(\mathbf{A})$. We therefore have $||\mathbf{A}|| \geq \rho(\mathbf{A})$.

Theorem:

$$\rho(\mathbf{A}) \leq ||\mathbf{A}^k||^{1/k} \tag{A.258}$$

Proof: Let λ and $\mathbf{v}$ by the eigenvalue and the corresponding eigenvector of $\mathbf{A}$ respectively, i.e.,

$$\mathbf{Av} = \lambda\mathbf{v}, \quad \text{and} \quad \mathbf{A}^k\mathbf{v} = \lambda^k\mathbf{v} \tag{A.259}$$

Taking norm on both sides we get

$$||\lambda^k\mathbf{v}|| = |\lambda|^k||\mathbf{v}|| = ||\mathbf{A}^k\mathbf{v}|| \leq ||\mathbf{A}^k||\,||\mathbf{v}|| \tag{A.260}$$

Dividing both sides by $||\mathbf{v}|| \neq 0$ we get

$$\lambda^k \leq ||\mathbf{A}^k||, \quad \text{i.e.} \quad \lambda \leq ||\mathbf{A}^k||^{1/k} \tag{A.261}$$

Theorem: A square matrix $\mathbf{A}$ is convergent, i.e., $\lim_{n\to\infty} \mathbf{A}^n = 0$, if and only if $\rho(\mathbf{A}) < 1$.

A.4.3 Vector and Matrix Differentiation

A multivariate function $f(\mathbf{x}) = f(x_1, \cdots, x_n)$ can be considered as a hypersurface in an $n+1$ dimensional space defined over an n-D space, i.e., corresponding to every point $\mathbf{x} = [x_1, \cdots, x_n]^T$ in the n-D space, there exists a unique value $f(\mathbf{x})$ (real or complex).

A *vector differentiation operator* is defined as

$$\frac{d}{d\mathbf{x}} = \left[\frac{\partial}{\partial x_1}, \cdots, \frac{\partial}{\partial x_n}\right]^T \tag{A.262}$$

which, when applied to any scalar function $f(\mathbf{x})$, produces its derivative with respect to $\mathbf{x}$, the *gradient* vector of $f(\mathbf{x})$:

$$\mathbf{g}_f(\mathbf{x}) = \bigtriangledown f(\mathbf{x}) = \frac{d}{d\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n}\right]^T = [f_{x_1}, \cdots, f_{x_n}]^T \tag{A.263}$$

The *Directional derivative* of a function $f(\mathbf{x})$ is defined as the rate of change of the function value along a unit direction $\mathbf{r} = [r_1, \cdots, r_n]^T$ (with $||\mathbf{r}|| = 1$):

$$D_r f(\mathbf{x}) = \bigtriangledown_r f(\mathbf{x}) = \frac{d}{dh} f(\mathbf{x} + h\mathbf{r}) = \lim_{h\to 0} \frac{f(\mathbf{x} + h\mathbf{r}) - f(\mathbf{x})}{h} \tag{A.264}$$

Introducing n intermediate functions $x_i + hr_i \;\; (i = 1, \cdots, n)$ of h, we can rewrite the above as the following by chain rule:

$$D_r f(\mathbf{x}) = \frac{d}{dh} f(\mathbf{x}+h\mathbf{r})\bigg|_{h=0} = \sum_{i=1}^{n} \frac{\partial f(\mathbf{x} + h\mathbf{r})}{\partial (x_i + hr_i)} \left.\frac{d(x_i + hr_i)}{dh}\right|_{h=0} = \sum_{i=1}^{n} f_{x_i}\, r_i = \mathbf{g}_f^T \mathbf{r} \tag{A.265}$$

We see that the directional derivative is the projection of the gradient vector onto the unit direction $\mathbf{r}$, which is maximized when the gradient vector $\mathbf{g}_f$ is in the same direction as $\mathbf{r}$. In other words, the function value increases mot rapidly along the direction of its gradient.

Vector differentiation has the following properties:

$$\frac{d}{d\mathbf{x}}(\mathbf{a}^T\mathbf{x}) = \frac{d}{d\mathbf{x}}(\mathbf{x}^T\mathbf{a}) = \mathbf{a}, \qquad \frac{d}{d\mathbf{x}}(\mathbf{x}^T\mathbf{x}) = 2\mathbf{x}, \qquad \frac{d}{d\mathbf{x}}(\mathbf{x}^T\mathbf{A}\mathbf{x}) = (\mathbf{A}^T + \mathbf{A})\mathbf{x} \tag{A.266}$$

To prove the third one, consider the kth element of the vector:

$$\frac{\partial}{\partial x_k}(\mathbf{x}^T\mathbf{A}\mathbf{x}) = \frac{\partial}{\partial x_k} \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j = \sum_{i=1}^{n} a_{ik}x_i + \sum_{j=1}^{n} a_{kj}x_j \qquad (k = 1, \cdots, n) \tag{A.267}$$

Putting all n elements in vector form, we have the above. If $\mathbf{A}^T = \mathbf{A}$ is symmetric, then we have

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^T\mathbf{A}\mathbf{x}) = 2\mathbf{A}\mathbf{x} \tag{A.268}$$

In particular, when $\mathbf{A} = \mathbf{I}$, we have

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^T\mathbf{x}) = 2\mathbf{x} \tag{A.269}$$

One can compare these results with the familiar derivatives in scalar case:

$$\frac{d}{dx}(ax^2) = 2ax \tag{A.270}$$

A *matrix differentiation operator* is defined as

$$\frac{d}{d\mathbf{A}} = \begin{bmatrix} \frac{\partial}{\partial a_{11}} & \cdots & \frac{\partial}{\partial a_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial a_{m1}} & \cdots & \frac{\partial}{\partial a_{mn}} \end{bmatrix} \tag{A.271}$$

which can be applied to any scalar function $f(\mathbf{A})$:

$$\frac{d}{d\mathbf{A}}f(\mathbf{A}) = \begin{bmatrix} \frac{\partial f(\mathbf{A})}{\partial a_{11}} & \cdots & \frac{\partial f(\mathbf{A})}{\partial a_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(\mathbf{A})}{\partial a_{m1}} & \cdots & \frac{\partial f(\mathbf{A})}{\partial a_{mn}} \end{bmatrix} \tag{A.272}$$

Specifically, consider $f(\mathbf{A}) = \mathbf{u}^T\mathbf{A}\mathbf{v}$, where $\mathbf{u}$ and $\mathbf{v}$ are $m \times 1$ and $n \times 1$ constant vectors, respectively, and $\mathbf{A}$ is an $m \times n$ matrix. Then we have:

$$\frac{d}{d\mathbf{A}}(\mathbf{u}^T\mathbf{A}\mathbf{v}) = \mathbf{u}\mathbf{v}^T \tag{A.273}$$

For more details refer to *Matrix Cookbook* at: `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook`

A.5 The Fundamental Theorem of Linear Algebra

A.5.1 Rank-Nullity Theorem

Definition: The number of independent rows in a matrix is called the row rank, the number of independent columns in a matrix is called the column rank.

Theorem: The row rank and column rank of any matrix are the same.

Proof: Let the column rank of the following $m \times n$ matrix $\mathbf{A}$ be r

$$\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_r, \mathbf{a}_{r+1}, \cdots, \mathbf{a}_n] = [\mathbf{A}_1,\ \mathbf{A}_0] \tag{A.274}$$

where $\mathbf{A}_1 = [\mathbf{a}_1, \cdots, \mathbf{a}_r]$ is an $m \times r$ sub-matrix formed by the r linearly independent columns in $\mathbf{A}$, assumed to be the first r columns (without loss of generality). Now consider the linear combination of another set of r n-D vectors $\{\mathbf{A}_1^T\mathbf{a}_1, \cdots, \mathbf{A}_1^T\mathbf{a}_r\}$:

$$\sum_{i=1}^{r} c_i\mathbf{A}_1^T\mathbf{a}_i = \mathbf{A}_1^T\sum_{i=1}^{r} c_i\mathbf{a}_i = \mathbf{A}_1^T\mathbf{A}_1\mathbf{c} = \mathbf{0} \tag{A.275}$$

which is assumed to be zero. As the $r \times r$ coefficient matrix $\mathbf{A}_1^T\mathbf{A}_1$ has full rank r ($\mathbf{A}_1$ is formed by r independent vectors), we must have $\mathbf{c} = [c_1, \cdots, c_r]^T = \mathbf{0}$,

indicating $\{\mathbf{A}_1^T\mathbf{a}_1, \cdots, \mathbf{A}_1^T\mathbf{a}_r\}$ are independent. However, as the linear combinations of r of the rows of $\mathbf{A}$, these r independent vector $\mathbf{A}_1^T\mathbf{a}_i$ $(i = 1, \cdots, r)$ are in the row space of $\mathbf{A}$. We therefore know that the row rank of $\mathbf{A}$ is no smaller than its column rank.

As the same argument can also be made for $\mathbf{A}^T$, i.e., the column rank of $\mathbf{A}$ is no smaller than its row rank, we conclude that the column and row ranks of $\mathbf{A}$ must be the same. QED

Rank-nullity theorem: The sum of the rank of an $m \times n$ matrix $\mathbf{A}$ and the dimension of its nullspace $N(\mathbf{A})$, the nullity of $\mathbf{A}$, is n:

$$\text{rank}(\mathbf{A}) + dim(N(\mathbf{A})) = \text{rank}(\mathbf{A}) + Nullity(\mathbf{A}) = n \tag{A.276}$$

Proof:

Let $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_r, \mathbf{a}_{r+1}, \cdots, \mathbf{a}_n]$ be an $m \times n$ matrix of rank r. We assume, without loss of generality, the first r columns are independent, and the last $n-r$ columns are dependent, and write it as $\mathbf{A} = [\mathbf{A}_1,\ \mathbf{A}_0]$. Here

$$\mathbf{A}_1 = [\mathbf{a}_1, \cdots, \mathbf{a}_r]_{m\times r} \tag{A.277}$$

is formed by the r independent columns of $\mathbf{A}$, i.e., if $\mathbf{A}_1\mathbf{x} = \mathbf{0}$, then $\mathbf{x} = \mathbf{0}$; and

$$\mathbf{A}_0 = [\mathbf{a}_{r+1}, \cdots, \mathbf{a}_n]_{m\times(n-r)} \tag{A.278}$$

is formed by the $n-r$ dependent columns of $\mathbf{A}$, i.e., any of them can be written as a linear combination of those independent columns $\mathbf{a}_1, \cdots, \mathbf{a}_r$ in $\mathbf{A}_1$:

$$\mathbf{a}_{r+k} = \sum_{i=1}^{r} c_{ik}\mathbf{a}_i = [\mathbf{a}_1, \cdots, \mathbf{a}_r]\begin{bmatrix} c_{1k} \\ \vdots \\ c_{rk} \end{bmatrix} = \mathbf{A}_1\mathbf{c}_k, \qquad (k = 1, \cdots, n-r) \tag{A.279}$$

where $\mathbf{c}_k = [c_{1k}, \cdots, c_{rk}]^T$ is a coefficient vector. We can put all these $n - r$ equations together to write them in matrix form as

$$\mathbf{A}_0 = [\mathbf{a}_{r+1}, \cdots, \mathbf{a}_n] = \mathbf{A}_1[\mathbf{c}_1, \cdots, \mathbf{c}_{n-r}] = \mathbf{A}_1\mathbf{C} \tag{A.280}$$

where $\mathbf{C} = [\mathbf{c}_1, \cdots, \mathbf{c}_{n-r}]$ is an $r \times (n-r)$ matrix composed of the $n-r$ coefficient vectors. The matrix $\mathbf{A}$ can now be written as

$$\mathbf{A} = [\mathbf{A}_1,\ \mathbf{A}_0] = [\mathbf{A}_1,\ \mathbf{A}_1\mathbf{C}] \tag{A.281}$$

Define an $n \times (n-r)$ matrix

$$\mathbf{X} = \begin{bmatrix} -\mathbf{C} \\ \mathbf{I} \end{bmatrix} = \begin{bmatrix} -\mathbf{c}_1 & \cdots & -\mathbf{c}_{n-r} \\ \mathbf{e}_1 & \cdots & \mathbf{e}_n \end{bmatrix} = \begin{bmatrix} -\mathbf{c}_1 & \cdots & -\mathbf{c}_{n-r} \\ 1 & \cdots & 0 \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \\ 0 & \cdots & 1 \end{bmatrix} \tag{A.282}$$

where $\mathbf{I}$ is an $(n-r) \times (n-r)$ identity matrix composed of n standard basis vectors $\mathbf{e}_1, \cdots, \mathbf{e}_n$. Now we have

$$\mathbf{AX} = [\mathbf{A}_1,\ \mathbf{A}_1\mathbf{C}] \begin{bmatrix} -\mathbf{C} \\ \mathbf{I} \end{bmatrix} = -\mathbf{A}_1\mathbf{C} + \mathbf{A}_1\mathbf{C} = \mathbf{O} \tag{A.283}$$

indicating any column $\mathbf{x}$ in $\mathbf{X}$ is a solution of the homogeneous equation $\mathbf{Ax} = \mathbf{0}$, i.e., $\mathbf{x}$ is in the nullspace $N(\mathbf{A})$. Also, if we let $\mathbf{Xu} = \mathbf{0}$, we get

$$\mathbf{Xu} = \begin{bmatrix} -\mathbf{C} \\ \mathbf{I} \end{bmatrix} \mathbf{u} = \begin{bmatrix} -\mathbf{Cu} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} = \mathbf{0} \tag{A.284}$$

i.e., $\mathbf{u} = \mathbf{0}$, indicating all columns in $\mathbf{X}$ are independent.

On the other hand, we can also show that any solution of the homogeneous equation $\mathbf{Ax} = \mathbf{0}$ must be a linear combination of the columns of $\mathbf{X}$. Assuming $\mathbf{x}$ is such a solution, i.e.,

$$\mathbf{Ax} = \mathbf{0} = [\mathbf{A}_1,\ \mathbf{A}_1\mathbf{C}] \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \mathbf{A}_1(\mathbf{x}_1 + \mathbf{C}\mathbf{x}_2) \tag{A.285}$$

As all columns in $\mathbf{A}$ are independent, we must have

$$\mathbf{x}_1 + \mathbf{C}\mathbf{x}_2 = \mathbf{0}, \quad \text{or} \quad \mathbf{x}_1 = -\mathbf{C}\mathbf{x}_2 \tag{A.286}$$

and we get

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{C}\mathbf{x}_2 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{C} \\ \mathbf{I} \end{bmatrix} \mathbf{x}_2 = \mathbf{X}\mathbf{x}_2 \tag{A.287}$$

indicating $\mathbf{x}$ is a linear combination of the $n-r$ columns in $\mathbf{X}$.

Based on the two results obtained above, we can now conclude that the nullspace $N(\mathbf{A})$ is spanned by the $n-r$ independent column vectors in $\mathbf{X}$, i.e., the dimension of $N(\mathbf{A})$ is $n-r$, and $rank(\mathbf{A}) + dim(\mathbf{A}) = n$.

A.5.2 Solving Linear Equation Systems

When solving a linear equation system $\mathbf{Ax} = \mathbf{b}$ with an $M \times N$ coefficient matrix $\mathbf{A}$, we need to answer some questions such as the following:

- Does a solution exist, i.e., can we find an $\mathbf{x}^*$ so that $\mathbf{Ax}^* = \mathbf{b}$ holds?
- If such a solution exists, is it unique? If it is not unique, how can we find all of the solutions?
- If no solution exists, can we still find the optimal approximated solution $\hat{\mathbf{x}}$ so that the error $||\mathbf{e}|| = ||\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}||$ is minimized?
- If the system has fewer equations than unknowns ($M < N$), are there infinite solutions?
- If the system has more equations than unknowns ($M > N$), is there no solution?

To answer all such questions, we consider the *fundamental theorem of linear algebra* to reveal the structure of the solutions of a given linear system $\mathbf{Ax} = \mathbf{b}$.

The $M \times N$ coefficient matrix $\mathbf{A} \in R^{M \times N}$ can be expressed in terms of either its N M-D column vectors $\mathbf{c}_j$ or its M N-D row vectors $\mathbf{r}_i^T$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \cdots & a_{MN} \end{bmatrix} = [\mathbf{c_1}, \cdots, \mathbf{c}_N] = \begin{bmatrix} \mathbf{r}_1^T \\ \vdots \\ \mathbf{r}_M^T \end{bmatrix}, \tag{A.288}$$

$$\mathbf{A}^T = \begin{bmatrix} a_{11} & \cdots & a_{M1} \\ \vdots & \ddots & \vdots \\ a_{1N} & \cdots & a_{MN} \end{bmatrix} = [\mathbf{r_1}, \cdots, \mathbf{r}_M] = \begin{bmatrix} \mathbf{c}_1^T \\ \vdots \\ \mathbf{c}_N^T \end{bmatrix} \tag{A.289}$$

where the ith row $\mathbf{r}_i$, $(i = 1, \cdots, M)$ and the jth column $\mathbf{c}_j$, $(j = 1, \cdots, N)$ of $\mathbf{A}$ are both represented as column vectors:

$$\mathbf{c}_j = \begin{bmatrix} a_{1j} \\ \vdots \\ a_{Mj} \end{bmatrix}, \qquad \mathbf{r}_i = \begin{bmatrix} a_{i1} \\ \vdots \\ a_{iN} \end{bmatrix} \quad \text{i.e.} \quad \mathbf{r}_i^T = [a_{i1}, \cdots, a_{iN}] \tag{A.290}$$

In general a function $y = f(x)$ can be represented by $f: X \to Y$, where

- X is the *domain* of the function, the set of all input or argument values;
- Y is the *codomain* of the function, the set into which all outputs of the function fall;
- The *image* of $f(x)$ is the set formed by the outputs of the function corresponding to all possible $x \in X$, a subset of the codomain.

A linear function $\mathbf{f}(\mathbf{x}) = \mathbf{Ax}$ maps an N-D vector $\mathbf{x} \in \mathbb{R}^N$ in the domain of the function into an M-D vector $\mathbf{Ax} \in R^M$ in the codomain of the function. The *fundamental theorem of linear algebra* concerns the following four subspaces all associated with with the $M \times N$ coefficient matrix $\mathbf{A}$ with rank $R = rank(\mathbf{A}) \leq \min(M, N)$, i.e., $\mathbf{A}$ has R independent columns and rows.

- The *column space* of $\mathbf{A}$ is a space spanned by its N M-D column vectors (of which $R \leq N$ are independent):

$$C(\mathbf{A}) = span(\mathbf{c}_1, \cdots, \mathbf{c}_N) \subseteq R^M \tag{A.291}$$

This is an R-D subspace of R^M composed of all possible linear combinations of its N column vectors:

$$x_1\mathbf{c}_1 + \cdots + x_N\mathbf{c}_N = [\mathbf{c_1}, \cdots, \mathbf{c}_N] \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} = \mathbf{Ax} \qquad (\text{for any } \mathbf{x} \in \mathbb{R}^N) \tag{A.292}$$

The column space $C(\mathbf{A})$ is the *image* of the linear transformation $\mathbf{f}(\mathbf{x}) =$

$\mathbf{Ax}$, and the equation $\mathbf{Ax} = \mathbf{b}$ is solvable if and only if $\mathbf{b} \in C(\mathbf{A})$. The dimension of the column space is the rank of $\mathbf{A}$, $dim\, C(\mathbf{A}) = rank(\mathbf{A}) = R$.

- The *row space* of $\mathbf{A}$ is a space spanned by its M N-D row vectors (of which $R \leq M$ are independent):

$$R(\mathbf{A}) = span(\mathbf{r}_1, \cdots, \mathbf{r}_M) \subseteq \mathbb{R}^N \tag{A.293}$$

This is an R-D subspace of $\mathbb{R}^N$ composed of all possible linear combinations of its M row vectors:

$$y_1\mathbf{r}_1 + \cdots + y_M\mathbf{r}_N = [\mathbf{r_1}, \cdots, \mathbf{r}_M] \begin{bmatrix} y_1 \\ \vdots \\ y_M \end{bmatrix} = \mathbf{A}^T\mathbf{y} \qquad (\text{for any } \mathbf{y} \in \mathbb{R}^M) \tag{A.294}$$

The row space $R(\mathbf{A})$ is the *image* of the linear transformation $\mathbf{f}(\mathbf{y}) = \mathbf{A}^T\mathbf{y}$, and the equation $\mathbf{A}^T\mathbf{y} = \mathbf{c}$ is solvable if and only if $\mathbf{c} \in R(\mathbf{A})$. As the rows and columns in $\mathbf{A}$ are respectively the columns and rows in $\mathbf{A}^T$, the row space of $\mathbf{A}$ is the column space of $\mathbf{A}^T$, and the column space of $\mathbf{A}$ is the row space of $\mathbf{A}^T$:

$$R(\mathbf{A}) = C(\mathbf{A}^T), \qquad C(\mathbf{A}) = R(\mathbf{A}^T) \tag{A.295}$$

The rank $R = rank(\mathbf{A})$ is the number of linearly independent rows and columns of $\mathbf{A}$, i.e., the row space and the column space have the same dimension, both equal to the rank of $\mathbf{A}$:

$$dim\, C(\mathbf{A}) = dim\, R(\mathbf{A}) = rank(\mathbf{A}) = R \tag{A.296}$$

- The *null space (kernel)* of $\mathbf{A}$, denoted by $N(\mathbf{A})$, is the set of all N-D vectors $\mathbf{x}$ that satisfy the homogeneous equation

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{r}_1^T \\ \vdots \\ \mathbf{r}_M^T \end{bmatrix} \mathbf{x} = \mathbf{0}, \qquad \text{or} \qquad \mathbf{r}_i^T\mathbf{x} = 0 \quad (i = 1, \cdots, M) \tag{A.297}$$

i.e.,

$$N(\mathbf{A}) = \{\mathbf{x} \in \mathbb{R}^N | \mathbf{Ax} = \mathbf{0}\} \subseteq \mathbb{R}^N \tag{A.298}$$

In particular, when $\mathbf{x} = \mathbf{0}$, we get $\mathbf{Ax} = \mathbf{0}$, i.e., the origin $\mathbf{0} \in N(\mathbf{A})$ is in the null space.

As $\mathbf{r}_i^T\mathbf{x} = 0$ for any $\mathbf{x} \in N(\mathbf{A})$ and $\mathbf{r}_i \in R(\mathbf{A})$, we see that the null space $N(\mathbf{A})$ and the row space $R(\mathbf{A}) = C(\mathbf{A}^T)$ are orthogonal to each other, $N(\mathbf{A}) \perp R(\mathbf{A})$.

The dimension of the null space is called the *nullity* of $\mathbf{A}$: $dim\ N(\mathbf{A}) = nullity(\mathbf{A})$. The rank-nullity theorem states the sum of the rank and the nullity of an $M \times N$ matrix $\mathbf{A}$ is equal to N:

$$\text{rank}(\mathbf{A}) + nullity(\mathbf{A}) = dim\ R(\mathbf{A}) + dim\ N(\mathbf{A}) = N \tag{A.299}$$

We therefore see that $R(\mathbf{A})$ and $N(\mathbf{A})$ are two mutually exclusive and complementary subspaces of $\mathbb{R}^N$:

$$R(\mathbf{A}) \perp N(\mathbf{A}), \quad R(\mathbf{A}) \cap N(\mathbf{A}) = \emptyset, \quad R(\mathbf{A}) \oplus N(\mathbf{A}) = \mathbb{R}^N \tag{A.300}$$

i.e., they are *orthogonal complement* of each other, denoted by

$$N(\mathbf{A}) = R(\mathbf{A})^\perp, \quad R(\mathbf{A}) = N(\mathbf{A})^\perp \tag{A.301}$$

Any N-D vector $\mathbf{x} \in \mathbb{R}^N$ is in either of the two subspaces $R(\mathbf{A})$ and $N(\mathbf{A})$.

- The null space of $\mathbf{A}^T$ (left null space of $\mathbf{A}$), denoted by $N(\mathbf{A}^T)$, is the set of all M-D vectors $\mathbf{y}$ that satisfy the homogeneous equation

$$\mathbf{A}^T\mathbf{y} = [\mathbf{c}_1, \cdots, \mathbf{c}_N]^T\mathbf{y} = \begin{bmatrix} \mathbf{c}_1^T \\ \vdots \\ \mathbf{c}_N^T \end{bmatrix} \mathbf{y} = \mathbf{0}, \quad \text{or} \quad \mathbf{c}_j^T\mathbf{y} = 0, \quad (j = 1, \cdots, N) \tag{A.302}$$

i.e.,

$$N(\mathbf{A}^T) = \{\mathbf{y} \in \mathbb{R}^M | \mathbf{A}^T\mathbf{y} = \mathbf{0}\} \subseteq \mathbb{R}^M \tag{A.303}$$

As all $\mathbf{y} \in N(\mathbf{A}^T)$ are orthogonal to $\mathbf{c}_i \in C(\mathbf{A})$, $N(\mathbf{A}^T)$ is orthogonal to the column space $C(\mathbf{A}) = R(\mathbf{A}^T)$:

$$N(\mathbf{A}^T) \perp C(\mathbf{A}), \qquad N(\mathbf{A}^T) \perp R(\mathbf{A}^T) \tag{A.304}$$

We see that $C(\mathbf{A})$ and $N(\mathbf{A}^T)$ are two mutually exclusive and complementary subspaces of $\mathbb{R}^M$:

$$\begin{aligned} &C(\mathbf{A}) \perp N(\mathbf{A}^T), \quad C(\mathbf{A}) \cap N(\mathbf{A}^T) = \emptyset, \quad C(\mathbf{A}) \oplus N(\mathbf{A}^T) = R^M, \\ &\text{i.e. } N(\mathbf{A}^T) = C(\mathbf{A})^\perp, \quad C(\mathbf{A}) = N(\mathbf{A}^T)^\perp \end{aligned} \tag{A.305}$$

Any M-D vector $\mathbf{y} \in \mathbb{R}^M$ is in either of the two subspaces $C(\mathbf{A})$ and $N(\mathbf{A}^T)$.

The four subspaces are summarized in the figure below, showing the domain $\mathbb{R}^N = R(\mathbf{A}) \oplus N(\mathbf{A})$ (left) and the codomain $\mathbb{R}^M = C(\mathbf{A}) \oplus N(\mathbf{A}^T)$ (right) of the linear mapping $\mathbf{Ax} = \mathbf{b}$, where

- $\mathbf{x}_p \in R(\mathbf{A})$ is the particular solution that is mapped to $\mathbf{Ax}_p = \mathbf{b} \in C(\mathbf{A})$, the image of $f(\mathbf{x}) = \mathbf{Ax}$;
- $\mathbf{x}_h \in N(\mathbf{A})$ is a homogeneous solution that is mapped to $\mathbf{Ax}_h = \mathbf{0} \in \mathbb{R}^M$;
- $\mathbf{x}_c = \mathbf{x}_p + \mathbf{x}_h$ is the complete solution that is mapped to $\mathbf{Ax}_c = \mathbf{b} \in C(\mathbf{A})$.

On the other hand, $\mathbf{y}_p \in C(\mathbf{A}) = R(\mathbf{A}^T)$, $\mathbf{y}_h \in N(\mathbf{A}^T)$, and $\mathbf{y}_c = \mathbf{y}_p + \mathbf{y}_h$ are respectively the particular, homogeneous and complete solutions of $\mathbf{A}^T\mathbf{y} = \mathbf{c}$. Here we have assumed $\mathbf{b} \in C(\mathbf{A})$ and $\mathbf{c} \in R(\mathbf{A})$, i.e., both $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{A}^T\mathbf{y} = \mathbf{c}$ are solvable. We will also consider the case where $\mathbf{b} \notin C(\mathbf{A})$ later.

We now consider specifically how to find the solutions of the system $\mathbf{Ax} = \mathbf{b}$ in light of the four subspaces of $\mathbf{A}$ defined above, through the examples below.

Example A.4 Solve the homogeneous equation system:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 2 & 5 \\ 2 & 3 & 8 \\ 3 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0} \tag{A.306}$$

We first convert $\mathbf{A}$ into the rref:

$$\begin{bmatrix} 1 & 2 & 5 \\ 2 & 3 & 8 \\ 3 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 5 \\ 0 & -1 & -2 \\ 0 & -5 & -10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 5 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$
$$\rightarrow \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{A.307}$$

The $R = rank(\mathbf{A}) = 2$ columns in the rref containing a single 1, called a *pivot*, are called the *pivot columns*, and the rows containing a pivot are called the *pivot rows*. Here, $R = 2 < M = N = 3$, i.e., $\mathbf{A}$ is a singular matrix. The two pivot rows $\mathbf{r}_1^T = [1,\ 0,\ 1]$ and $\mathbf{r}_2^T = [0,\ 1,\ 2]$ can be used as the basis vectors that span the row space $R(\mathbf{A})$:

$$R(\mathbf{A}) = c_1\mathbf{r}_1 + c_2\mathbf{r}_2 = c_1 \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + c_2 \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \tag{A.308}$$

Note that the pivot columns of the rref do not span the column space $C(\mathbf{A})$, as the row reduction operations do not reserve the columns of $\mathbf{A}$. But they indicate the corresponding columns $\mathbf{c}_1 = [1\ 2\ 3]^T$ and $\mathbf{c}_2 = [2\ 3\ 1]^T$ in the original matrix $\mathbf{A}$ can be used as the basis that spans $C(\mathbf{A})$. In general the bases of the row and column spaces so obtained are not orthogonal.

The R pivot rows are the independent equations in the system of M equations, and the variables corresponding to the pivot columns (here x_1 and x_2) are the *pivot variables*. The remaining $M - R$ non-pivot rows containing all zeros are not independent, and the variables corresponding to the non-pivot rows are *free variables* (here x_3), which can take any values.

From the rref form of the equation, we get

$$x_1 + x_3 = 0, \quad x_2 + 2x_3 = 0 \tag{A.309}$$

If we let the free variable x_3 take the value 1, then we can get the two pivot variables $x_1 = -1$ and $x_2 = -2$, and a special homogeneous solution $\mathbf{x}_h = [-1\ -2\ 1]^T$ as a basis vector that spans the 1-D null space $N(\mathbf{A})$. However, as the free variable x_3 can take any value c, the complete solution is the entire 1-D null space:

$$N(\mathbf{A}) = c\mathbf{x}_h = c \begin{bmatrix} -1 \\ -2 \\ 1 \end{bmatrix} \tag{A.310}$$

Example A.5 Solve the non-homogeneous equation with the same coefficient matrix $\mathbf{A}$ used in the previous example:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 2 & 5 \\ 2 & 3 & 8 \\ 3 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 19 \\ 31 \\ 22 \end{bmatrix} = \mathbf{b} \tag{A.311}$$

We use Gauss-Jordan elimination to solve this system:

$$\begin{aligned} [\ \mathbf{A} \quad \mathbf{b}\ ] &= \left[\begin{array}{ccc|c} 1 & 2 & 5 & 19 \\ 2 & 3 & 8 & 31 \\ 3 & 1 & 5 & 22 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 5 & 19 \\ 0 & -1 & -2 & -7 \\ 0 & -5 & -10 & -35 \end{array}\right] \\ &\rightarrow \left[\begin{array}{ccc|c} 1 & 2 & 5 & 19 \\ 0 & 1 & 2 & 7 \\ 0 & 0 & 0 & 0 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 0 & 1 & 5 \\ 0 & 1 & 2 & 7 \\ 0 & 0 & 0 & 0 \end{array}\right] \end{aligned} \tag{A.312}$$

The $R = 2$ pivot rows correspond to the independent equations in the system, i.e., $rand(\mathbf{A}) = 2$, while the remaining $M - R = 1$ non-pivot row does not play any role as they map any $\mathbf{x}$ to $\mathbf{0}$. As $\mathbf{A}$ is singular, $\mathbf{A}^{-1}$ does not exist. However, we can find the solution based on the rref of the system, which can also be expressed in block matrix form:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 0 \end{bmatrix} \qquad \text{or} \qquad \begin{bmatrix} \mathbf{I} & \mathbf{F} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{pivot} \\ \mathbf{x}_{free} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{0} \end{bmatrix} \tag{A.313}$$

where

$$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{x}_{pivot} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{x}_{free} = x_3, \quad \mathbf{b}_1 = \begin{bmatrix} 5 \\ 7 \end{bmatrix} \tag{A.314}$$

Solving the matrix equation above for $\mathbf{x}_{pivot}$, we get

$$\mathbf{I}\mathbf{x}_{pivot} + \mathbf{F}\mathbf{x}_{free} = \mathbf{b}_1, \qquad \mathbf{x}_{pivot} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \mathbf{b}_1 - \mathbf{F}\mathbf{x}_{free} = \begin{bmatrix} 5 \\ 7 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ -2 \end{bmatrix} \tag{A.315}$$

If we let $\mathbf{x}_{free} = x_3 = 0$, we get a particular solution $\mathbf{x}_p = [5\ 7\ 0]^T$, which can be expressed as a linear combination of $\mathbf{r}_1$ and $\mathbf{r}_2$ that span $R(\mathbf{A})$, and $\mathbf{x}_h$ that span $N(\mathbf{A})$:

$$\mathbf{x}_p = \begin{bmatrix} 5 \\ 7 \\ 0 \end{bmatrix} = \frac{11}{6}\mathbf{r}_1 + \frac{2}{3}\mathbf{r}_2 - \frac{19}{6}\mathbf{x}_h = \frac{11}{6} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \frac{2}{3} \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} - \frac{19}{6} \begin{bmatrix} -1 \\ -2 \\ 1 \end{bmatrix} \tag{A.316}$$

We see that this solution is not entirely in the row space $R(\mathbf{A})$. In general, this is the case for all particular solutions so obtained.

Having found both the particular solution $\mathbf{x}_p$ and the homogeneous solution $\mathbf{x}_h$, we can further find the complete solution $\mathbf{x}_c$ as the sum of $\mathbf{x}_p$ and the entire

null space spanned by $\mathbf{x}_h$:

$$\mathbf{x}_c = \begin{bmatrix} 5 \\ 7 \\ 0 \end{bmatrix} + c \begin{bmatrix} -1 \\ -2 \\ 1 \end{bmatrix} = \mathbf{x}_p + c\mathbf{x}_h = \mathbf{x}_p + N(\mathbf{A}) \tag{A.317}$$

Based on different constant c, we get a set of equally valid solutions. For example, if $c = 0, 1, 2$, then we get

$$\mathbf{x}_0 = \begin{bmatrix} 5 \\ 7 \\ 0 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{bmatrix} 4 \\ 5 \\ 1 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 3 \\ 3 \\ 2 \end{bmatrix} \tag{A.318}$$

These solutions have the same projection onto the row space $R(\mathbf{A})$, i.e., they have the same projections onto the two basis vectors $\mathbf{r}_1$ and $\mathbf{r}_2$ that span $R(\mathbf{A})$:

$$\mathbf{x}_0^T\mathbf{r}_1 = \mathbf{x}_1^T\mathbf{r}_1 = \mathbf{x}_2^T\mathbf{r}_1 = 5, \qquad \mathbf{x}_0^T\mathbf{r}_2 = \mathbf{x}_1^T\mathbf{r}_2 = \mathbf{x}_2^T\mathbf{r}_2 = 7 \tag{A.319}$$

The figure below shows how the complete solution $\mathbf{x}_c$ can be obtained as the sum of a particular solution $\mathbf{x}_p$ in $R(\mathbf{A})$ and the entire null space $N(\mathbf{A})$. Here $N = 3$ and space $\mathbb{R}^3$ is composed of $R(\mathbf{A})$ and $N(\mathbf{A})$, respectively 2-D and 1-D on the left, but 1-D and 2-D on the right. In either case, the complete solution is any particular solution plus the entire null space, the vertical dashed line on the left, the top dashed plane on the right. All points on the vertical line or top satisfy the equation system, as they are have the same projection onto the row space $R(\mathbf{A})$.

If the right hand side is $\mathbf{b}' = [19, 31, 20]^T$, then the rref of the equation becomes:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 2 \end{bmatrix} \tag{A.320}$$

The non-pivot row is an impossible equation $0 = 2$, indicating that no solution exists, as $\mathbf{b}' \notin C(\mathbf{A})$ is not in the column space spanned by $\mathbf{r}_1 = [1\ 0\ 1]^T$ and $\mathbf{r}_2 = [0\ 1\ 2]^T$.

Example A.6 Find the complete solution of the following linear equation system:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ -2 & 1 & 4 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix} = \mathbf{b} \tag{A.321}$$

This equation can be solved in the following steps:

- Construct the augmented matrix and then convert it to the rref form:

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 3 \\ 4 & 3 & 2 & 1 & 2 \\ -2 & 1 & 4 & 7 & 4 \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 3 \\ 0 & -5 & -10 & -15 & -10 \\ 0 & 5 & 10 & 15 & 10 \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 3 \\ 0 & 1 & 2 & 3 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$
$$\rightarrow \left[\begin{array}{cccc|c} 1 & 0 & -1 & -2 & -1 \\ 0 & 1 & 2 & 3 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] = \left[\begin{array}{cc|c} \mathbf{I} & \mathbf{F} & \mathbf{b_1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right]$$

The two pivot rows $\mathbf{r}_1^T = [1\ 0\ -1\ -2]$ and $\mathbf{r}_2^T = [0\ 1\ 2\ 3]$ in the rref span $R(\mathbf{A})$, and the two columns in the *original* matrix $\mathbf{A}$ corresponding to the pivot columns in the rref, $[1\ 4\ -2]^T$ and $[2\ 3\ 1]^T$ could be used as the basis vectors that span $C(\mathbf{A})$.

The equation system can be represented in block matrix form:

$$\left[\begin{array}{cc} \mathbf{I} & \mathbf{F} \\ \mathbf{0} & \mathbf{0} \end{array}\right] \left[\begin{array}{c} \mathbf{x}_p \\ \mathbf{x}_f \end{array}\right] = \left[\begin{array}{c} \mathbf{b_1} \\ \mathbf{0} \end{array}\right] \tag{A.322}$$

where

$$\mathbf{I} = \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right], \quad \mathbf{F} = \left[\begin{array}{cc} -1 & -2 \\ 2 & 3 \end{array}\right], \quad \mathbf{x}_p = \left[\begin{array}{c} x_1 \\ x_2 \end{array}\right], \quad \mathbf{x}_f = \left[\begin{array}{c} x_3 \\ x_4 \end{array}\right], \quad \mathbf{b}_1 = \left[\begin{array}{c} -1 \\ 2 \end{array}\right] \tag{A.323}$$

Multiplying out we get

$$\mathbf{x}_p + \mathbf{F}\mathbf{x}_f = \mathbf{b}_1 \tag{A.324}$$

- Find the homogeneous solution for equation $\mathbf{A}\mathbf{x} = \mathbf{0}$ by setting $\mathbf{b}_1 = \mathbf{0}$:

$$\mathbf{I}\mathbf{x}_p + \mathbf{F}\mathbf{x}_f = \mathbf{0}, \quad \text{i.e.} \quad \mathbf{x}_p = \left[\begin{array}{c} x_1 \\ x_2 \end{array}\right] = -\mathbf{F}\mathbf{x}_f = \left[\begin{array}{cc} 1 & 2 \\ -2 & -3 \end{array}\right] \left[\begin{array}{c} x_3 \\ x_4 \end{array}\right] \tag{A.325}$$

We let $\mathbf{x}_f$ be either of the two standard basis vectors $\mathbf{x}_f = [x_3,\ x_4]^T = [1\ 0]^T$ and $\mathbf{x}_f = [x_3,\ x_4]^T = [0\ 1]^T$ of the null space $N(\mathbf{A})$, and get

$$\mathbf{x}_p = \left[\begin{array}{c} x_1 \\ x_2 \end{array}\right] = \left[\begin{array}{c} 1 \\ -2 \end{array}\right] \quad \text{or} \quad \left[\begin{array}{c} 2 \\ -3 \end{array}\right] \tag{A.326}$$

and the two corresponding homogeneous solutions:

$$\mathbf{x}_{h1} = \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}\right] = \left[\begin{array}{c} 1 \\ -2 \\ 1 \\ 0 \end{array}\right], \qquad \mathbf{x}_{h2} = \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}\right] = \left[\begin{array}{c} 2 \\ -3 \\ 0 \\ 1 \end{array}\right] \tag{A.327}$$

- Find the particular solution of the non-homogeneous equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ by

setting $\mathbf{x}_f = \mathbf{0}$

$$\mathbf{I}\mathbf{x}_p + \mathbf{F}\mathbf{x}_f = \mathbf{x}_p = \mathbf{b}_1, \quad \text{i.e.} \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \quad \text{i.e.} \quad \mathbf{x}_p = \begin{bmatrix} -1 \\ 2 \\ 0 \\ 0 \end{bmatrix} \tag{A.328}$$

- Find the complete solution:

$$\mathbf{x}_c = \mathbf{x}_p + N(\mathbf{A}) = \mathbf{x}_p + c_1\mathbf{x}_{h1} + c_2\mathbf{x}_{h2} = \begin{bmatrix} -1 \\ 2 \\ 0 \\ 0 \end{bmatrix} + c_1 \begin{bmatrix} 1 \\ -2 \\ 1 \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 2 \\ -3 \\ 0 \\ 1 \end{bmatrix} \tag{A.329}$$

If the right-hand side is $\mathbf{b}' = [1,\ 3,\ 5]^T$, then the row reduction of the augmented matrix yields:

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 1 \\ 4 & 3 & 2 & 1 & 3 \\ -2 & 1 & 4 & 7 & 5 \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 3 \\ 0 & -5 & -10 & -15 & -1 \\ 0 & 5 & 10 & 15 & 7 \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 1 \\ 0 & 1 & 2 & 3 & 0.2 \\ 0 & 0 & 0 & 0 & 1.2 \end{array}\right] \tag{A.330}$$

The equation corresponding to the last non-pivot row is $0 = 1.2$, indicating the system is not solvable (even though the coefficient matrix does not have full rank), because $\mathbf{b}' \notin C(\mathbf{A})$ is not in the column space.

Example A.7 Consider the linear equation system with a coefficient matrix $\mathbf{A}^T$, the transpose of $\mathbf{A}$ used in the previous example:

$$\mathbf{A}^T\mathbf{y} = \begin{bmatrix} 1 & 4 & -2 \\ 2 & 3 & 1 \\ 3 & 2 & 4 \\ 4 & 1 & 7 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 11 \\ 19 \\ 27 \end{bmatrix} = \mathbf{c} \tag{A.331}$$

- Convert the augmented matrix into the rref form:

$$\left[\begin{array}{ccc|c} 1 & 4 & -2 & 3 \\ 2 & 3 & 1 & 11 \\ 3 & 2 & 4 & 19 \\ 4 & 1 & 7 & 27 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 4 & -2 & 3 \\ 0 & -5 & 5 & 5 \\ 0 & -10 & 10 & 10 \\ 0 & -15 & 15 & 15 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 4 & -2 & 3 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 0 & 2 & 7 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \tag{A.332}$$

The two pivot rows $\mathbf{r}_1^T = [1\ 0\ 2]$ and $\mathbf{r}_2^T = [0\ 1\ -1]$ are the basis vectors that span $R(\mathbf{A}^T) = C(\mathbf{A})$. The two vectors that span $C(\mathbf{A})$ found in Example A.6 can be expressed as linear combinations of $\mathbf{r}_1$ and $\mathbf{r}_2$, $[1\ 4\ -2]^T = \mathbf{r}_1 + 4\mathbf{r}_2$ and $[2\ 3\ 1]^T = 2\mathbf{r}_1 + 3\mathbf{r}_2$, i.e., either of the two pairs can be used as the basis that span $C(\mathbf{A})$.

Based on the rref above, the equation system can now be written as

$$\begin{bmatrix} \mathbf{I} & \mathbf{F} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{y}_p \\ \mathbf{y}_f \end{bmatrix} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{0} \end{bmatrix} \tag{A.333}$$

where

$$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \quad \mathbf{y}_p = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \mathbf{y}_f = y_3, \quad \mathbf{c}_1 = \begin{bmatrix} 7 \\ -1 \end{bmatrix} \tag{A.334}$$

Multiplying out we get

$$\mathbf{I}\mathbf{y}_p + \mathbf{F}\mathbf{y}_f = \mathbf{c}_1 \tag{A.335}$$

- Find the homogeneous solution for $\mathbf{A}^T\mathbf{y} = \mathbf{c} = \mathbf{0}$ by setting $\mathbf{c} = \mathbf{0}$ and thereby $\mathbf{c}_1 = \mathbf{0}$. We let the free variable $\mathbf{y}_f = y_3 = 1$ and get

$$\mathbf{I}\mathbf{y}_p + \mathbf{F}\mathbf{y}_f = \mathbf{0}, \quad \text{i.e.} \quad \mathbf{y}_p = -\mathbf{F}\mathbf{y}_f = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = -\begin{bmatrix} 2 \\ -1 \end{bmatrix} y_3 = \begin{bmatrix} -2 \\ 1 \end{bmatrix},$$

$$\text{i.e.,} \quad \mathbf{y}_h = \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \tag{A.336}$$

- Find the particular solution of the non-homogeneous equation $\mathbf{A}^T\mathbf{y} = \mathbf{c} \neq \mathbf{0}$ by setting $\mathbf{y}_f = y_3 = 0$:

$$\mathbf{I}\mathbf{y}_p + \mathbf{F}\mathbf{y}_f = \mathbf{y}_p = \mathbf{c}_1 = \begin{bmatrix} 7 \\ -1 \end{bmatrix}, \quad \text{i.e.} \quad \mathbf{y}_p = \begin{bmatrix} 7 \\ -1 \\ 0 \end{bmatrix} \tag{A.337}$$

- Find the complete solution:

$$\mathbf{y}_c = \mathbf{y}_p + N(\mathbf{A}^T) = \mathbf{y}_p + cy_3 = \begin{bmatrix} 7 \\ -1 \\ 0 \end{bmatrix} + c \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \tag{A.338}$$

If the right-hand side is $\mathbf{c}' = [1,\ 1,\ 2,\ 3]^T$,

$$\left[\begin{array}{ccc|c} 1 & 4 & -2 & 1 \\ 2 & 3 & 1 & 1 \\ 3 & 2 & 4 & 2 \\ 4 & 1 & 7 & 3 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 4 & -2 & 1 \\ 0 & -5 & 5 & -1 \\ 0 & -10 & 10 & -1 \\ 0 & -15 & 15 & -1 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 4 & -2 & 1 \\ 0 & 1 & -1 & 1/5 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 \end{array}\right] \tag{A.339}$$

indicating the system is not solvable, as this $\mathbf{c}' \notin C(\mathbf{A}^T) = R(\mathbf{A})$, i.e., it is not in the column space of $\mathbf{A}^T$ or row space of $\mathbf{A}$.

In these examples, we have obtained all four subspaces associated with this matrix $\mathbf{A}$ with $M = 3$, $N = 4$, and $R = 2$, in terms of the bases that span the subspaces:

1. The row space $R(\mathbf{A}) = C(\mathbf{A}^T)$ is an R-D subspace of $\mathbb{R}^4$, spanned by the $R = 2$ pivot rows of the rref of $\mathbf{A}$:

$$R(\mathbf{A}) = C(\mathbf{A}^T) = span\left(\begin{bmatrix} 1 \\ 0 \\ -1 \\ -2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}\right) \tag{A.340}$$

2. The null space $N(\mathbf{A})$ is an (N-R)-D subspace of $\mathbb{R}^4$ spanned by the $N - R$ independent homogeneous solutions:

$$N(\mathbf{A}) = span\left(\begin{bmatrix} 1 \\ -2 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ -3 \\ 0 \\ 1 \end{bmatrix}\right) \tag{A.341}$$

   Note that the basis vectors of $N(\mathbf{A})$ are indeed orthogonal to those of $R(\mathbf{A})$.
3. The column space of $\mathbf{A}$ is the same as the row space of $\mathbf{A}^T$, which is the R-D subspace of R^3 spanned by the two pivot rows of the rref of $\mathbf{A}^T$.

$$C(\mathbf{A}) = R(\mathbf{A}^T) = span\left(\begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}\right) \tag{A.342}$$

4. The left null space $N(\mathbf{A}^T)$ is a (M-R)-D subspace of $\mathbb{R}^3$ spanned by the homogeneous solutions (here one $3 - 2 = 1$ solution):

$$N(\mathbf{A}^T) = span\left(\begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}\right) = c \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} \tag{A.343}$$

   Again note that the basis vectors of $N(\mathbf{A}^T)$ are orthogonal to those of $C(\mathbf{A})$.

In general, here are the ways to find the bases of the four subspaces:

- The basis vectors of $R(\mathbf{A})$ are the pivot rows of the rref of $\mathbf{A}$.
- The basis vectors of $C(\mathbf{A})$ are the pivot rows of the rref of $\mathbf{A}^T$.
- The basis vectors of $N(\mathbf{A})$ are the independent homogeneous solutions of $\mathbf{Ax} = \mathbf{0}$. To find them, reduce $\mathbf{A}$ to the rref, identify all free variables $\mathbf{x}_f$ corresponding to non-pivot columns, set one of them to 1 and the rest to 0, solve homogeneous system $\mathbf{Ax} = \mathbf{0}$ to find the pivot variables $\mathbf{x}_p$ to get one basis vector. Repeat the process for each of the free variables to get all basis vectors.
- The basis vectors of $R(\mathbf{A}^T)$ can be obtained by doing the same as above for $\mathbf{A}^T$.

Note that while the basis of $R(\mathbf{A})$ are the pivot rows of the rref of $\mathbf{A}$, as its rows are equivalent to those of $\mathbf{A}$, the pivot columns of the rref basis of $C(\mathbf{A})$ are not the basis of $C(\mathbf{A})$, as the columns of $\mathbf{A}$ have been changed by the row deduction

operations and are therefore not equivalent to the columns of the resulting rref. The columns in $\mathbf{A}$ corresponding to the pivot columns in the rref could be used as the basis of $C(\mathbf{A})$. Alternatively, the basis of $C(\mathbf{A})$ can be obtained from the rref of $\mathbf{A}^T$, as its rows are equivalent to those of $\mathbf{A}^T$, which are the columns of $\mathbf{A}$.

We further make the following observations:

- The basis vectors of each of the four subspaces are independent, the basis vectors of $R(\mathbf{A})$ and $N(\mathbf{A})$ are orthogonal, and $dim\ R(\mathbf{A}) + dim\ N(\mathbf{A}) = M$. Similarly, the basis vectors of $C(\mathbf{A})$ and $N(\mathbf{A}^T)$ are orthogonal, and $dim\ C(\mathbf{A}) + dim\ N(\mathbf{A}^T) = N$. In other words, the four subspaces indeed satisfy the following orthogonal and complementary properties:

$$C(\mathbf{A}) \perp N(\mathbf{A}^T), \quad R(\mathbf{A}) \perp N(\mathbf{A}), \quad C(\mathbf{A}) \oplus N(\mathbf{A}^T) = R^3, \quad R(\mathbf{A}) \oplus N(\mathbf{A}) = R^4 \tag{A.344}$$

  i.e., they are orthogonal complements: $C(\mathbf{A}) = N(\mathbf{A})^\perp, \quad R(\mathbf{A}) = N(\mathbf{A})^\perp$.

- For $\mathbf{Ax} = \mathbf{b}$ to be solvable, the constant vector $\mathbf{b}$ on the right-hand side must be in the column space, $\mathbf{b} \in C(\mathbf{A})$. Otherwise the equation is not solvable, even if $R < M < N$. Similarly, for $\mathbf{A}^T\mathbf{x} = \mathbf{c}$ to be solvable, $\mathbf{c}$ must be in the row space $\mathbf{c} \in R(\mathbf{A})$. In the examples above, both $\mathbf{b}$ and $\mathbf{c}$ are indeed in their corresponding column spaces:

$$\begin{aligned} \mathbf{b} &= \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix} = 3 \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} + 2 \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \in C(\mathbf{A}) \\ \mathbf{c} &= \begin{bmatrix} 3 \\ 11 \\ 19 \\ 27 \end{bmatrix} = 3 \begin{bmatrix} 1 \\ 0 \\ -1 \\ -2 \end{bmatrix} + 11 \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix} \in R(\mathbf{A}) \end{aligned} \tag{A.345}$$

  But as $\mathbf{b} = [1,\ 3,\ 5]^T \notin C(\mathbf{A})$ and $\mathbf{c} = [1,\ 1,\ 2,\ 3]^T \notin R(\mathbf{A})$, the corresponding systems have no solutions.

- All homogeneous solutions of $\mathbf{Ax} = \mathbf{0}$ are in the null space $\mathbf{x}_h \in N(\mathbf{A})$, but in general the particular solutions $\mathbf{x}_p \in R^N = R(\mathbf{A}) \oplus N(\mathbf{A})$ are not necessarily in the row space $R(\mathbf{A})$. In the examples above, $\mathbf{x}_p \in R^N$ is a linear combination of the $R = 2$ basis vectors of $R(\mathbf{A})$ and $N - R = 2$ basis vectors of $N(\mathbf{A})$:

$$\mathbf{x}_p = \begin{bmatrix} -1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \left(0.1 \begin{bmatrix} 1 \\ 0 \\ -1 \\ -2 \end{bmatrix} + 0.2 \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix} \right) + \left(-0.3 \begin{bmatrix} 1 \\ -2 \\ 1 \\ 0 \end{bmatrix} - 0.4 \begin{bmatrix} 2 \\ -3 \\ 0 \\ 1 \end{bmatrix} \right) = \mathbf{x}'_p + \mathbf{x}''_p \tag{A.346}$$

where

$$\mathbf{x}'_p = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix} \in R(\mathbf{A}), \qquad \mathbf{x}''_p = \begin{bmatrix} -1.1 \\ 1.8 \\ -0.3 \\ -0.4 \end{bmatrix} \in N(\mathbf{A}) \tag{A.347}$$

are the projections of $\mathbf{x}_p$ onto $R(\mathbf{A})$ and $N(\mathbf{A})$, respectively, and $\mathbf{x}'_p$ is another particular solution without any homogeneous component that satisfies $\mathbf{A}\mathbf{x} = \mathbf{b}$, while $\mathbf{x}''_p$ is a homogeneous solution satisfying $\mathbf{A}\mathbf{x} = \mathbf{0}$.

- All homogeneous solutions of $\mathbf{A}^T\mathbf{y} = \mathbf{c}$ are in the left null space $\mathbf{y}_h \in N(\mathbf{A}^T)$, but in general the particular solutions $\mathbf{y}_p \in C(\mathbf{A}) \oplus N(\mathbf{A}^T) = R^M$ are not necessarily in the column space. In the examples above, $\mathbf{y}_p \in R^M$ is a linear combination of the $R = 2$ basis vectors of $C(\mathbf{A})$ and $M - R = 1$ basis vector of $N(\mathbf{A}^T)$:

$$\mathbf{y}_p = \begin{bmatrix} 7 \\ -1 \\ 0 \end{bmatrix} = \left(2 \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} + 1.5 \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \right) - 2.5 \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1.5 \\ 2.5 \end{bmatrix} - 2.5 \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix} = \mathbf{y}'_p + \mathbf{y}''_p \tag{A.348}$$

where $\mathbf{y}'_p = [2,\ 1.5,\ 2.5]^T \in C(\mathbf{A})$ is a particular solution (without any homogeneous component) that satisfies $\mathbf{A}^T\mathbf{y} = \mathbf{c}$.

Here is a summary of the four subspaces associated with an M by N matrix $\mathbf{A}$ of rank R.

	$dimR(\mathbf{A})$	$dimN(\mathbf{A})$	$dimC(\mathbf{A})$	$dimR(\mathbf{A}^T)$	solvability of $\mathbf{A}\mathbf{x} = \mathbf{b}$
$M = N = R$	R	0	R	0	unique solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$
$M > N = R$	R	0	R	$M - R$	over-constrained, solvable if $\mathbf{b} \in C(\mathbf{A})$
$N > M = R$	R	$N - R$	R	0	under-constrained, infinite solutions $\mathbf{x} =$
$R < min(M, N)$	R	$N - R$	R	$M - R$	solvable if $\mathbf{b} \in C(\mathbf{A})$, infinite solutions

(A.349)

The figure below illustrates a specific case with $M = N = 3$ and $R = 2$. As $\mathbf{b} \notin C(\mathbf{A})$, the system can only be approximately solved to find $\mathbf{A}\mathbf{x} = \mathbf{p} \in C(\mathbf{A})$, which is the projection of $\mathbf{b}$ onto the column space $C(\mathbf{A})$. The error $||\mathbf{e}|| = ||\mathbf{p} - \mathbf{b}|| = ||\mathbf{A}\mathbf{x}_p - \mathbf{b}||$ is minimized, $\mathbf{x}_p$ is the optimal approximation. We will consider ways to obtain this optimal approximation in the following sections.

A.6 Pseudo-Inverse

A.6.1 Pseudo-Inverse

The inverse of an $m \times n$ matrix $\mathbf{A}$ does not exist if it is not square $m \neq n$. But we can still find its *pseudo-inverse*, an $n \times m$ matrix denoted by $\mathbf{A}^-$, if $rank(\mathbf{A}) = r = \min(m, n)$, in either of the following ways:

- If $m > n = r$, $\mathbf{A}^T\mathbf{A}$ is an $n \times n$ full-rank invertible matrix, and we define the *left inverse*:

$$\mathbf{A}^- = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \tag{A.350}$$

which satisfies

$$\mathbf{A}^-\mathbf{A} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{A} = \mathbf{I}_{n\times n} \tag{A.351}$$

But

$$\mathbf{A}\mathbf{A}^- = \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \neq \mathbf{I} \tag{A.352}$$

- If $n > m = r$, $\mathbf{A}\mathbf{A}^T$ is an $m \times m$ full-rank invertible matrix, and we define the *right inverse*:

$$\mathbf{A}^- = \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1} \tag{A.353}$$

which satisfies

$$\mathbf{A}\mathbf{A}^- = \mathbf{A}\mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1} = \mathbf{I}_{m\times m} \tag{A.354}$$

But

$$\mathbf{A}^-\mathbf{A} = \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1}\mathbf{A} \neq \mathbf{I} \tag{A.355}$$

The left and right inverses defined respectively for $m > n$ and $m < n$ are the same. If $m > n$, then we have the left inverse $\mathbf{A}^- = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$. We then define $\mathbf{B} = \mathbf{A}^T$, and take the transpose on $\mathbf{A}^-$ to get

$$(\mathbf{A}^-)^T = [(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T]^T = \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1} = (\mathbf{A}^T)^-; \tag{A.356}$$

i.e., $\mathbf{B}^- = \mathbf{B}^T(\mathbf{B}\mathbf{B}^T)^{-1}$, which is the second definition of the pseudo inverse of $\mathbf{B}$.

We can show that $(\mathbf{A}^-)^- = \mathbf{A}$. If $m > n$, then we have

$$\begin{aligned}(\mathbf{A}^-)^- &= [(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T]^- = [(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T]^T\left[(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T[(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T]^T\right]^{-1}\\ &= \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\left[(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\right]^{-1}\\ &= \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}(\mathbf{A}^T\mathbf{A}) = \mathbf{A}.\end{aligned}$$

Similarly we can show this is the case if $m < n$.

In particular, when $m = n$, $\mathbf{A}$ is a square invertible matrix, and both the left and right pseudo-inverse defined above become the regular inverse

$$\begin{aligned}\mathbf{A}^- &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T = \mathbf{A}^{-1}(\mathbf{A}^T)^{-1}\mathbf{A}^T = \mathbf{A}^{-1}\\ \mathbf{A}^- &= \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1} = \mathbf{A}^T(\mathbf{A}^T)^{-1}\mathbf{A}^{-1} + \mathbf{A}^{-1}\end{aligned}$$

If $r < \min(m, n)$, then $\mathbf{A}^{-1}$ does not exist, and $\mathbf{A}^-$ is neither left nor right inverse, because $\mathbf{A}^-\mathbf{A} \neq \mathbf{I}$ and $\mathbf{A}\mathbf{A}^- \neq \mathbf{I}$, as they are not full rank.

The pseudo inverse of $\mathbf{A}$ can also be obtained based on its SVD $\mathbf{A}_{m\times n} = \mathbf{U}_{m\times m}\mathbf{\Sigma}_{m\times n}\mathbf{V}^T_{n\times n}$. Consider the following two cases:

- If $m > n$, the left pseudo inverse is

$$\begin{aligned}\mathbf{A}^- &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T = [(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)^T(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)]^{-1}(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)^T \\ &= (\mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)^{-1}(\mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T) = \mathbf{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\mathbf{V}^T\mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T \\ &= \mathbf{V}[(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma})^{-1}\boldsymbol{\Sigma}^T]\mathbf{U}^T = \mathbf{V}\boldsymbol{\Sigma}^-\mathbf{U}^T\end{aligned}$$

- If $m < n$, the right pseudo inverse is

$$\begin{aligned}\mathbf{A}^- &= \mathbf{A}^T(\mathbf{A}\mathbf{A}^T)^{-1} = (\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)^T[(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T)^T]^{-1} \\ &= (\mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T)(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T\mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T)^{-1} = \mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T(\mathbf{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^T\mathbf{U}^T)^{-1} \\ &= \mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T\mathbf{U}(\boldsymbol{\Sigma}\boldsymbol{\Sigma}^T)^{-1}\mathbf{U}^T = \mathbf{V}[\boldsymbol{\Sigma}^T(\boldsymbol{\Sigma}\boldsymbol{\Sigma}^T)^{-1}]\mathbf{U}^T = \mathbf{V}\boldsymbol{\Sigma}^-\mathbf{U}^T\end{aligned}$$

In either case, we have

$$\mathbf{A}^-_{n\times m} = (\mathbf{U}_{m\times m}\boldsymbol{\Sigma}_{m\times m}\mathbf{V}^T_{n\times n})^- = \mathbf{V}_{n\times n}\boldsymbol{\Sigma}^-_{n\times m}\mathbf{U}^T_{m\times m} \tag{A.357}$$

where $\boldsymbol{\Sigma}^-_{n\times m}$ is the pseudo inverse of $\boldsymbol{\Sigma}_{m\times n}$ with $1/\sigma_1, \cdots, 1/\sigma_r$ on its diagonal.

We also have

$$\mathbf{A}^-\mathbf{A} = (\mathbf{V}\boldsymbol{\Sigma}^-\mathbf{U}^T)(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T) = \mathbf{V}(\boldsymbol{\Sigma}^-\boldsymbol{\Sigma})\mathbf{V}^T \tag{A.358}$$

if $r = m$, then $\boldsymbol{\Sigma}^-\boldsymbol{\Sigma} = \mathbf{I}$ an $n \times n$ identity matrix and we have $\mathbf{A}^-\mathbf{A} = \mathbf{V}\mathbf{V}^T = \mathbf{I}$. Otherwise if $r < m$, $\boldsymbol{\Sigma}^-\boldsymbol{\Sigma}$ has zero components on the diagonal, and $\mathbf{A}^-\mathbf{A} \neq \mathbf{I}$.

A.6.2 Pseudo-Inverse Solutions Based on SVD

In the previous section we obtained the solution of the equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ together with the bases of the four subspaces of $\mathbf{A}$ based its rref. Here we will consider an alternative and better way to solve the same equation and find a set of orthogonal bases that also span the four subspaces, based on the pseudo-inverse and the singular value decomposition (SVD) of $\mathbf{A}$. The solution obtained this way is optimal in some certain sense as shown below.

Consider the SVD of an $M \times N$ matrix $\mathbf{A}$ of rank $R \leq \min(M, N)$:

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T \tag{A.359}$$

where

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1 & & & & & 0 \\ & \ddots & & & & \\ & & \sigma_R & & & \\ & & & 0 & & \\ & & & & \ddots & \\ 0 & & & & & 0 \end{bmatrix}_{M\times N} \tag{A.360}$$

is an $M \times N$ matrix with R non-zero *singular values* $\sigma_i = \sqrt{\lambda_i}$ $(i = 1, \ldots, R)$ of $\mathbf{A}$ along the diagonal (starting from the top-left corner), while all other components are zero, and $\mathbf{U} = [\mathbf{u}_1, \cdots, \mathbf{u}_M]$ and $\mathbf{V} = [\mathbf{v}_1, \cdots, \mathbf{v}_N]$ are two orthogonal

matrices of dimensions $M \times M$ and $N \times N$ respectively. The column vectors $\mathbf{u}_i$ $(i = 1, \cdots, M)$ and $\mathbf{v}_j$ $(j = 1, \cdots, N)$, are called the left and right *singular vectors* of $\mathbf{A}$, respectively, and they can be used as the orthonormal bases to span respectively $\mathbb{R}^M = C(\mathbf{A}) \oplus N(\mathbf{A}^T)$ and its subspaces $C(\mathbf{A})$ and $N(\mathbf{A}^T)$, and $\mathbb{R}^N = C(\mathbf{A}) \oplus N(\mathbf{A})$ and its subspaces $C(\mathbf{A})$ and $N(\mathbf{A})$.

To see this, we rewrite the SVD of $\mathbf{A}$ as:

$$\begin{aligned}
\mathbf{A} &= [\mathbf{c}_1, \cdots, \mathbf{c}_N] = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \\
&= [\mathbf{u}_1, \cdots\cdots\cdots, \mathbf{u}_M] \begin{bmatrix} \sigma_1 & & & & & 0 \\ & \ddots & & & & \\ & & \sigma_R & & & \\ & & & 0 & & \\ & & & & \ddots & \\ 0 & & & & & 0 \end{bmatrix}_{M\times N} \begin{bmatrix} \mathbf{v}_1^T \\ \vdots \\ \vdots \\ \vdots \\ \mathbf{v}_N^T \end{bmatrix} \\
&= \sum_{k=1}^{R} \sigma_k \ \left(\mathbf{u}_k \mathbf{v}_k^T\right) = \sum_{k=1}^{R} \sigma_k \ \mathbf{u}_k [v_{1k}, \cdots, v_{Nk}] \\
&= \left[\sum_{k=1}^{R} (\sigma_k v_{1k}) \ \mathbf{u}_k, \cdots, \sum_{k=1}^{R} (\sigma_k v_{Nk}) \ \mathbf{u}_k \right]
\end{aligned} \tag{A.361}$$

Each column vector of $\mathbf{A}$ can be expressed as a linear combination of the first R columns of $\mathbf{U}$ corresponding to the non-zero singular values:

$$\mathbf{c}_j = \sum_{k=1}^{R} (\sigma_k v_{jk}) \ \mathbf{u}_k \qquad (j = 1, \cdots, N), \tag{A.362}$$

Taking transpose on both sides of the SVD we also get:

$$\begin{aligned}
\left(\mathbf{A}^T\right) &= [\mathbf{r}_1, \cdots, \mathbf{r}_M] = \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T \\
&= \sum_{k=1}^{R} \sigma_k \ \left(\mathbf{v}_k \mathbf{u}_k^T\right) = \sum_{k=1}^{R} \sigma_k \ \mathbf{v}_k [u_{1k}, \cdots, u_{Mk}] \\
&= \left[\sum_{k=1}^{R} (\sigma_k u_{1k}) \ \mathbf{v}_k, \ \cdots, \ \sum_{k=1}^{R} (\sigma_k u_{Mk}) \ \mathbf{v}_k \right]
\end{aligned} \tag{A.363}$$

i.e., each row vector of $\mathbf{A}$ can be expressed as a linear combination of the first R columns of $\mathbf{V}$ corresponding to the non-zero singular values:

$$\mathbf{r}_i = \sum_{k=1}^{R} (\sigma_k u_{ik}) \ \mathbf{v}_k \qquad (i = 1, \cdots, M) \tag{A.364}$$

We therefore see that the R columns of $\mathbf{U}$ and $\mathbf{V}$ corresponding to the non-zero singular values span respectively the column space $C(\mathbf{A})$ and row space $R(\mathbf{A})$:

$$C(\mathbf{A}) = span(\mathbf{u}_1, \cdots, \mathbf{u}_R), \qquad R(\mathbf{A}) = span(\mathbf{v}_1, \cdots, \mathbf{v}_R) \tag{A.365}$$

and the remaining $M - R$ columns of $\mathbf{U}$ and $N - R$ columns of $\mathbf{V}$ corresponding

to the zero singular values span respectively $N(\mathbf{A}^T)$ orthogonal to $C(\mathbf{A})$, and $N(\mathbf{A})$ orthogonal to $R(\mathbf{A})$:

$$N(\mathbf{A}^T) = span(\mathbf{u}_{R+1}, \cdots, \mathbf{u}_M), \qquad N(\mathbf{A}) = span(\mathbf{v}_{R+1}, \cdots, \mathbf{v}_N) \tag{A.366}$$

Subspace	Definition	Dimension	Basis
$C(\mathbf{A}) \subseteq R^M$	column space (image) of $\mathbf{A}$	R	Rcolumns of$\mathbf{U}$corresponding to non-zero
$N(\mathbf{A}^\mathbf{T}) \subseteq R^M$	left null space of $\mathbf{A}^T$	$M-R$	$M-R$columns of$\mathbf{U}$corresponding to zer
$R(\mathbf{A}) \subseteq \mathbb{R}^N$	row space (image) of $\mathbf{A}^T$	R	Rcolumns of$\mathbf{V}$corresponding to non-zero
$N(\mathbf{A}) \subseteq \mathbb{R}^N$	null space of $\mathbf{A}$	$N-R$	$N-R$columns of $\mathbf{V}$ corresponding to zer
$\mathbf{R}^N = N(\mathbf{A}) \oplus R(\mathbf{A})$	domain of $\mathbf{A}$	N	all N columns of $\mathbf{V}$
$\mathbf{R}^M = N(\mathbf{A}^T) \oplus C(\mathbf{A})$	codomain of $\mathbf{A}$	M	all M columns of $\mathbf{U}$

(A.367)

The SVD method can be used to find the pseudo-inverse of an $M \times N$ matrix $\mathbf{A}$ of rank $R \leq min(M, N)$:

$$\mathbf{A}^- = \mathbf{V}\boldsymbol{\Sigma}^-\mathbf{U}^T \tag{A.368}$$

where both $\mathbf{A}^-$ and

$$\boldsymbol{\Sigma}^- = \begin{bmatrix} 1/\sigma_1 & & & & & 0 \\ & \ddots & & & & \\ & & 1/\sigma_R & & & \\ & & & 0 & & \\ & & & & \ddots & \\ 0 & & & & & 0 \end{bmatrix}_{N \times M} \tag{A.369}$$

are $N \times M$ matrices.

We further note that matrices $\mathbf{U}$ and $\mathbf{V}$ are related to each other by:

$$\begin{cases} \mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T \\ \mathbf{A}^- = \mathbf{V}\boldsymbol{\Sigma}^-\mathbf{U}^T \end{cases} \quad \text{or} \quad \begin{cases} \mathbf{AV} = \mathbf{U}\boldsymbol{\Sigma} \\ \mathbf{A}^-\mathbf{U} = \mathbf{V}\boldsymbol{\Sigma}^- \end{cases} \tag{A.370}$$

or in vector form:

$$\begin{cases} \mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i & (i = 1, \cdots, R) \\ \mathbf{A}^-\mathbf{u}_i = \mathbf{v}_i/\sigma_i & (i = 1, \cdots, R) \end{cases} \qquad \begin{cases} \mathbf{A}\mathbf{v_i} = 0 & (i = R+1, \cdots, N) \\ \mathbf{A}^-\mathbf{u_i} = 0 & (i = R+1, \cdots, M) \end{cases} \tag{A.371}$$

indicating how the individual columns are related. We see that the last $N - R$ columns $\mathbf{v}_{R+1}, \cdots, \mathbf{v}_N$ of $\mathbf{V}$ form an orthogonal basis of $N(\mathbf{A})$; and the last $M - R$ columns $\mathbf{u}_{R+1}, \cdots, \mathbf{u}_M$ of $\mathbf{U}$ form an orthogonal basis of $N(\mathbf{A}^T)$.

We now show that the optimal solution of the linear system $\mathbf{Ax} = \mathbf{b}$ can be obtained based on the pseudo-inverse of $\mathbf{A}$:

$$\mathbf{x}_{svd} = \mathbf{A}^-\mathbf{b} = \mathbf{V}\boldsymbol{\Sigma}^-\mathbf{U}^T\mathbf{b} \tag{A.372}$$

Specially, if $M = N = R$, then $\mathbf{A}^- = \mathbf{A}^{-1}$, and $\mathbf{x}_{svd} = \mathbf{A}^{-1}\mathbf{b}$ is the unique and exact solution. In general when $M \neq N$ or $R < \min(M, N)$, the solution

may not exist, or it may not be unique, but $\mathbf{x}_{svd}$ is still an optimal solution in two ways, from the perspective of both the domain and codomain of the linear mapping $\mathbf{Ax}$, as shown below.

- In domain $\mathbb{R}^N$:

  Pre-multiplying $\mathbf{V}^T = \mathbf{V}^{-1}$ on both sides of the pseudo-inverse solution $\mathbf{x}_{svd} = \mathbf{V}\mathbf{\Sigma}^-\mathbf{U}^T\mathbf{b}$ given above, we get:

  $$\mathbf{V}^T\mathbf{x}_{svd} = [\mathbf{v}_1, \cdots, \mathbf{v}_N]^T\mathbf{x}_{svd} = \mathbf{\Sigma}^-\mathbf{U}^T\mathbf{b} = \mathbf{\Sigma}^-[\mathbf{u}_1, \cdots, \mathbf{u}_M]^T\mathbf{b}, \quad \text{(A.373)}$$

  or in component form:

  $$\begin{bmatrix} \mathbf{v}_1^T\mathbf{x}_{svd} \\ \vdots \\ \mathbf{v}_R^T\mathbf{x}_{svd} \\ \mathbf{v}_{R+1}^T\mathbf{x}_{svd} \\ \vdots \\ \mathbf{v}_N^T\mathbf{x}_{svd} \end{bmatrix} = \begin{bmatrix} 1/\sigma_1 & & & & & 0 \\ & \ddots & & & & \\ & & 1/\sigma_R & & & \\ & & & 0 & & \\ & & & & \ddots & \\ 0 & & & & & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1^T\mathbf{b} \\ \vdots \\ \mathbf{u}_R^T\mathbf{b} \\ \mathbf{u}_{R+1}^T\mathbf{b} \\ \vdots \\ \mathbf{u}_M^T\mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1^T\mathbf{b}/\sigma_1 \\ \vdots \\ \mathbf{u}_R^T\mathbf{b}/\sigma_R \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{(A.374)}$$

  The first R components $\mathbf{v}_i^T\mathbf{x}_{svd} = \mathbf{u}_i^T\mathbf{b}/\sigma_i$ $(i = 1, \cdots, R)$ are the projection of $\mathbf{x}_{svd}$ onto $R(\mathbf{A})$ spanned by $\{\mathbf{v}_1, \cdots, \mathbf{v}_R\}$ corresponding to the non-zero singular values, while the last $N - R$ components $\mathbf{v}_i^T\mathbf{x}_{svd} = 0$ $(i = R + 1, \cdots, N)$ are the projection of $\mathbf{x}_{svd}$ onto $N(\mathbf{A})$ spanned by $\{\mathbf{v}_{R+1}, \cdots, \mathbf{v}_N\}$ corresponding to the zero singular values. We see that the pseudo-inverse solution $\mathbf{x}_{svd}$ is entirely in $R(\mathbf{A})$, containing no homogeneous component in $N(\mathbf{A})$. In other words, $\mathbf{x}_{svd}$ has the minimum norm (closest to the origin) compared to any other possible solution $\mathbf{x}_p$, such as those found previously based on the rref of $\mathbf{A}$, containing a non-zero homogeneous component $\mathbf{x}_h \in N(\mathbf{A})$:

  $$||\mathbf{x}_{svd}|| \leq ||\mathbf{x}_p|| = ||\mathbf{x}_{svd} + \mathbf{x}_h|| \quad \text{(A.375)}$$

  As $N(\mathbf{A}) \perp R(\mathbf{A})$, $\mathbf{x}_{svd} \in R(\mathbf{A})$ is the projection of any such solution $\mathbf{x}_p$ onto $R(\mathbf{A})$. The complete solution can be found as

  $$\mathbf{x}_c = \mathbf{x}_{svd} + N(\mathbf{A}) = \mathbf{x}_{svd} + \sum_{j=R+1}^{N} c_j\mathbf{v}_j \quad \text{(A.376)}$$

  for any set of coefficients c_j.

- In codomain $\mathbb{R}^M$:

  We now consider the result $\mathbf{b}_{svd} = \mathbf{A}\mathbf{x}_{svd}$ produced by the pseudo-inverse solution $\mathbf{x}_{svd}$, which, as a linear combination of the columns of $\mathbf{A}$, is in its column space $C(\mathbf{A})$:

  $$\mathbf{b}_{svd} = \mathbf{A}\mathbf{x}_{svd} = \mathbf{A}\mathbf{A}^-\mathbf{b} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}^-\mathbf{U}^T\mathbf{b} = \mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^-\mathbf{U}^T\mathbf{b} \quad \text{(A.377)}$$

  Pre-multiplying $\mathbf{U}^T = \mathbf{U}^{-1}$ on both sides we get:

  $$\mathbf{U}^T\mathbf{b}_{svd} = \mathbf{\Sigma}\mathbf{\Sigma}^-\mathbf{U}^T\mathbf{b} \quad \text{(A.378)}$$

or in component form:

$$\begin{bmatrix} \mathbf{u}_1^T\mathbf{b}_{svd} \\ \vdots \\ \mathbf{u}_R^T\mathbf{b}_{svd} \\ \mathbf{u}_{R+1}^T\mathbf{b}_{svd} \\ \vdots \\ \mathbf{u}_M^T\mathbf{b}_{svd} \end{bmatrix} = \begin{bmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & & 0 & & \\ & & & & \ddots & \\ 0 & & & & & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1^T\mathbf{b} \\ \vdots \\ \mathbf{u}_R^T\mathbf{b} \\ \mathbf{u}_{R+1}^T\mathbf{b} \\ \vdots \\ \mathbf{u}_M^T\mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1^T\mathbf{b} \\ \vdots \\ \mathbf{u}_R^T\mathbf{b} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{A.379}$$

The first R components $\mathbf{u}_i^T\mathbf{b}_{svd} = \mathbf{u}_i^T\mathbf{b}$ $(i = 1, \cdots, R)$ are the projection of $\mathbf{b}_{svd}$ onto $C(\mathbf{A})$ spanned by $\{\mathbf{u}_1, \cdots, \mathbf{u}_R\}$ corresponding to the non-zero singular values, while the last $M - R$ components $\mathbf{u}_i^T\mathbf{b}_{svd} = 0$ $(i = R + 1, \cdots, M)$ are the projection of $\mathbf{b}_{svd}$ onto $N(\mathbf{A}^T)$ spanned by $\{\mathbf{u}_{R+1}, \cdots, \mathbf{u}_M\}$ corresponding to the zero singular values. We see that $\mathbf{b}_{svd}$ and $\mathbf{b}$ have the same projection onto $C(\mathbf{A})$. If $\mathbf{b} \in C(\mathbf{A})$, i.e., $\mathbf{u}_i^T\mathbf{b} = 0$ for all $i = R+1, \cdots, M$, then $\mathbf{b}_{svd} = \mathbf{A}\mathbf{x}_{svd} = \mathbf{b}$ and $\mathbf{x}_{svd}$ is an exact solution. But if $\mathbf{b} \notin C(\mathbf{A})$, as it contains non-zero components in $N(\mathbf{A}^T)$, then no solution exists. However, as $N(\mathbf{A}^T) \perp C(\mathbf{A})$, $\mathbf{b}_{svd} \in C(\mathbf{A})$ is the projection of $\mathbf{b}$ onto $C(\mathbf{A})$, the error $||\mathbf{b} - \mathbf{b}_{svd}||$ is minimized when compared with that of any other approximate solution $\mathbf{x}_p \neq \mathbf{x}_{svd}$:

$$||\mathbf{b} - \mathbf{b}_{svd}|| = ||\mathbf{b} - \mathbf{A}\mathbf{x}_{svd}|| \leq ||\mathbf{b} - \mathbf{A}\mathbf{x}_p|| \tag{A.380}$$

In this sense, $\mathbf{x}_{svd}$ is the optimal.

Summarizing the two aspects above, we see that the pseudo-inverse solution $\mathbf{x}_{svd}$ is optimal in the sense that both its norm $||\mathbf{x}_{svd}||$ and its error $||\mathbf{A}\mathbf{x}_{svd} - \mathbf{b}||$ are minimized.

- If the solution $\mathbf{x}_{svd}$ is not unique because $N(\mathbf{A}) \neq \emptyset$, then the complete solution can be found by adding the entire null space to it: $\mathbf{x}_c = \mathbf{x}_{svd} + N(\mathbf{A})$.
- If no solution exists because $\mathbf{b} \notin C(\mathbf{A})$, then $\mathbf{x}_{svd}$ is the optimal approximate solution with minimum error.

Example A.8 Given the same system considered in previous examples

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ -2 & 1 & 4 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix} \tag{A.381}$$

we will now solve it using the pseudo-inverse method. We first find SVD of $\mathbf{A}$ in terms of the following matrices:

$$\mathbf{U} = [\mathbf{u}_1,\ \mathbf{u}_2,\ \mathbf{u}_3] = \begin{bmatrix} -0.535 & -0.218 & 0.817 \\ -0.267 & -0.873 & -0.408 \\ -0.802 & 0.436 & -0.408 \end{bmatrix} \tag{A.382}$$

$$\mathbf{V} = [\mathbf{v}_1,\ \mathbf{v}_2,\ \mathbf{v}_3,\ \mathbf{v}_4] = \begin{bmatrix} 0.000 & -0.837 & 0.374 & -0.400 \\ -0.267 & -0.478 & -0.797 & 0.255 \\ -0.535 & -0.120 & 0.472 & 0.691 \\ -0.802 & 0.239 & -0.049 & -0.546 \end{bmatrix} \tag{A.383}$$

$$\mathbf{\Sigma} = \begin{bmatrix} 10.0 & 0 & 0 & 0 \\ 0 & 5.477 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{\Sigma}^- = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.183 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{A.384}$$

$$\mathbf{A}^- = \mathbf{V}\mathbf{\Sigma}^-\mathbf{U}^T = \begin{bmatrix} 0.000 & -0.837 & 0.374 & -0.400 \\ -0.267 & -0.478 & -0.797 & 0.255 \\ -0.535 & -0.120 & 0.472 & 0.691 \\ -0.802 & 0.239 & -0.049 & -0.546 \end{bmatrix} \begin{bmatrix} 0.100 & 0 & 0 \\ 0 & 0.183 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.535 & -0.2 \\ -0.218 & -0.8 \\ 0.817 & -0.4 \end{bmatrix}$$

$$= \begin{bmatrix} 0.033 & 0.133 & -0.067 \\ 0.033 & 0.083 & -0.017 \\ 0.033 & 0.033 & 0.033 \\ 0.033 & -0.017 & 0.083 \end{bmatrix}$$

The particular solution of the system $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{b} = [3,\ 2,\ 4]^T \in C(\mathbf{A})$ is

$$\mathbf{x}_{svd} = \mathbf{A}^-\mathbf{b} = \begin{bmatrix} 0.033 & 0.133 & -0.067 \\ 0.033 & 0.083 & -0.017 \\ 0.033 & 0.033 & 0.033 \\ 0.033 & -0.017 & 0.083 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix}$$

$$= -0.535 \begin{bmatrix} 0.000 \\ -0.267 \\ -0.535 \\ -0.802 \end{bmatrix} - 0.120 \begin{bmatrix} -0.837 \\ -0.478 \\ -0.120 \\ 0.239 \end{bmatrix} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2$$

which is in $R(\mathbf{A})$ spanned by the first $R = 2$ columns $\mathbf{v}_1$ and $\mathbf{v}_2$, perpendicular to $N(\mathbf{A})$ spanned by the last $N - R = 2$ columns $\mathbf{v}_3$ and $\mathbf{v}_4$. Note that this solution $\mathbf{x}_{svd} = [0.1,\ 0.2,\ 0.3,\ 0.4]^T$ is actually the first component $\mathbf{x}'_p \in R(\mathbf{A})$ of the particular solution $\mathbf{x}_p = [-1,\ 2,\ 0,\ 0]$ found in the previous section by Gauss-Jordan elimination, which is not in $R(\mathbf{A})$. Adding the null space to the particular solution, we get the complete solution:

$$\mathbf{x}_c = \mathbf{x}_{svd} + N(\mathbf{A}) = \mathbf{x}_p + c_1\mathbf{v}_3 + c_2\mathbf{v}_4 = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix} + c_1 \begin{bmatrix} 0.374 \\ -0.797 \\ 0.472 \\ -0.049 \end{bmatrix} + c_2 \begin{bmatrix} -0.400 \\ 0.255 \\ 0.691 \\ -0.546 \end{bmatrix} \tag{A.385}$$

If the right-hand side is replaced by $\mathbf{b} = [1,\ 3,\ 5]^T \notin C(\mathbf{A})$, no solution exists. However, we can still find the pseudo-inverse solution as the optimal approximate

solution:

$$\mathbf{x}_{svd} = \mathbf{A}^{-}\mathbf{b} = \begin{bmatrix} 0.033 & 0.133 & -0.067 \\ 0.033 & 0.083 & -0.017 \\ 0.033 & 0.033 & 0.033 \\ 0.033 & -0.017 & 0.083 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix} \tag{A.386}$$

which is the same as the solution for $\mathbf{b} = [3,\ 2,\ 4]^T$, indicating $[3,\ 2,\ 4]^T$ happens to be the projection of $[1,\ 3,\ 5]^T$ onto $C(\mathbf{A})$ spanned by $\mathbf{u}_1$ and $\mathbf{u}_2$. The result produced by $\mathbf{x}_{svd}$ is $\mathbf{b}' = \mathbf{A}\mathbf{x}_{svd} \in C(\mathbf{A})$ is the projection of $\mathbf{b}$ onto $C(\mathbf{A})$, with a minimum error distance $||\mathbf{e}|| = ||\mathbf{b} - \mathbf{b}'||$, indicating $\mathbf{x}_{svd}$ is the optimal approximate solution.

Homework 3:

1. Use Matlab function `svd(A)` to carry out the SVD of the coefficient matrix $\mathbf{A}$ of the linear system in Homework 2 problem 3,

$$\mathbf{A} = \begin{bmatrix} -1 & 3 & 4 & 1 \\ 2 & -4 & 3 & 2 \\ 1 & -1 & 7 & 3 \end{bmatrix} \tag{A.387}$$

   Find $\mathbf{U}$, $\mathbf{V}$, $\mathbf{\Sigma}$. Verify that $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, i.e., $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}$.
2. Obtain $\mathbf{\Sigma}^{-}$, i.e, find the transpose of $\mathbf{\Sigma}$ and then replace each singular value by its reciprocal, verify that $\mathbf{\Sigma}\mathbf{\Sigma}^{-} = \mathbf{I}$ and $\mathbf{\Sigma}^{-}\mathbf{\Sigma} = \mathbf{I}$. Then find the pseudo-inverse $\mathbf{A}^{-} = \mathbf{U}\mathbf{\Sigma}^{-}\mathbf{V}^T$.
3. Use Matlab function `pinv(A)` to find the pseudo-inverse $\mathbf{\Sigma}^{-}$ and $\mathbf{A}^{-}$. Compare them to what you obtained in the previous part.
4. Identify the bases of the four subspaces $R(\mathbf{A})$, $N(\mathbf{A})$, $C(\mathbf{A}) = R(\mathbf{A}^T)$, and $N(\mathbf{A}^T)$ based on $\mathbf{U}$ and $\mathbf{V}$. Verify that these bases and those you found previously (problem 3 of Homework 2) span the same spaces, i.e., the basis vectors of one basis can be written as a linear combination of those of the other, for each of the four subspaces.
5. Use the pseudo-inverse $\mathbf{A}^{-}$ found above to solve the system $\mathbf{A}\mathbf{x} = \mathbf{b}$. Find the two particular solutions $\mathbf{x}_{p1}$ and $\mathbf{x}_{p2}$ corresponding to two different right-hand side $\mathbf{b}_1 = [1,\ 2,\ 3]^T$ and $\mathbf{b}_2 = [2,\ 3,\ 2]^T$.
6. How are the two results $\mathbf{x}_{p1}$ and $\mathbf{x}_{p2}$ related to each other? Give your explanation. Why can't you find a solution when $\mathbf{b} = [2,\ 3,\ 2]^T$ but SVD method can?
7. Verify that $\mathbf{x}_p \perp N(\mathbf{A})$. Also find the error (or residual) $\mathbf{e} = \mathbf{b} - \mathbf{A}\mathbf{x}_p$ for each of the two results above. Verify that $\mathbf{e} \perp C(\mathbf{A})$, if $\mathbf{e} \neq 0$.
8. In Homework 2 you used row reduction method to solve the system $\mathbf{A}\mathbf{x} = \mathbf{b}_1$ and you should have found a particular solution $\mathbf{x}_1 = [5,\ 2,\ 0,\ 0,\ ]^T$. Also it is obvious to see that another solution is $\mathbf{x}_2 = [0,\ 0,\ 0,\ 1]^T$. Show that the projection of these solutions onto $R(\mathbf{A})$ spanned by the first two columns of $\mathbf{V}$ is the same as the particular solution $\mathbf{x}_p$ found by the SVD method.

9. Show that $\mathbf{b}_1 = [1,\ 2,\ 3]^T$ is the projection of $\mathbf{b}_2 = [2,\ 3,\ 2]^T$ onto $C(\mathbf{A})$, the 2-D subspace spanned by the first two columns of $\mathbf{U}$. Can you also show that it is the projection of $\mathbf{b}$ onto $C(\mathbf{A})$ spanned by the basis you obtained in Homework 2 (not necessarily orthogonal)?
10. Give an expression of the null space $N(\mathbf{A})$, and then write the complete solution in form of $\mathbf{x}_c = \mathbf{x}_p + N(\mathbf{A})$. Verify any complete solution so generated satisfies $\mathbf{A}\mathbf{x}_c = \mathbf{b}_1$.

A.7 Taylor Series Expansion

- **Single-variable functions:**

  A single-variable function $f(x)$ can be expanded around a given point x by the Taylor series:

$$f(x+\delta x) = f(x) + f'(x)\delta x + \frac{1}{2!}f''(x)\delta x^2 + \cdots + \frac{1}{n!}f^{(n)}(x)\delta x^n + O(\delta x^{n+1}) \tag{A.388}$$

  where $O(\delta x^{n+1})$ denotes all the remaining terms with order higher than n in the infinite series. If δx is much smaller than 1, $O(\delta x^{n+1})$ can be neglected so that the function $f(x)$ can be approximated by the first n terms. For example, $f(x)$ can be approximated as a quadratic function with a third order error term $O(\delta x^3)$ when $n = 2$, or a linear function with a second order error term $O(\delta x^2)$ when $n = 1$:

$$\begin{aligned} f(x+\delta x) &= f(x) + f'(x)\delta x + \frac{1}{2!}f''(x)\delta x^2 + O(\delta x^3) \\ &\approx f(x) + f'(x)\delta x + \frac{1}{2!}f''(x)\delta x^2 \\ &\approx f(x) + f'(x)\delta x \end{aligned} \tag{A.389}$$

- **Multi-variable scalar-valued functions:**

  A multi-variable function $f(x_1, \cdots, x_n) = f(\mathbf{x})$ can also be expanded by the Taylor series:

$$\begin{aligned} f(\mathbf{x}+\delta\mathbf{x}) = f(\mathbf{x}) &+ \sum_{j=1}^{n}\frac{\partial f(\mathbf{x})}{\partial x_j}\delta x_j + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\delta x_i \delta x_j \\ &+ \frac{1}{3!}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\frac{\partial^3 f(\mathbf{x})}{\partial x_i \partial x_j \partial x_k}\delta x_i \delta x_j \delta x_k + \cdots \end{aligned} \tag{A.390}$$

  which can be expressed in vector form as:

$$f(\mathbf{x}+\delta\mathbf{x}) = f(\mathbf{x}) + \mathbf{g}_f^T\,\delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^T\mathbf{H}_f\,\delta\mathbf{x} + \cdots \tag{A.391}$$

  where $\delta\mathbf{x} = [\delta x_1, \cdots, \delta x_n]^T$, while $\mathbf{g}_f$ and $\mathbf{H}_f$ are the gradient vector and

the *Hessian* matrix for the first and second order derivatives of the function, respectively:

$$\mathbf{g}_f(\mathbf{x}) = \bigtriangledown f(\mathbf{x}) = \frac{d}{d\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f\mathbf{x})}{\partial x_n} \end{bmatrix}_{n\times 1}, \tag{A.392}$$

$$\mathbf{H}_f(\mathbf{x}) = \frac{d}{d\mathbf{x}}\mathbf{g}_f(\mathbf{x}) = \frac{d^2}{d\mathbf{x}^2} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}_{n\times n} = \mathbf{H}_f^T \tag{A.393}$$

Note that $\mathbf{H}_f^T$ is symmetric as $\partial^2 f(\mathbf{x})/\partial x_i \partial x_j = \partial^2 f(\mathbf{x})/\partial x_j \partial x_i$.

When $\delta\mathbf{x}$ is small (i.e., $||\delta\mathbf{x}||$ is small), then $f(\mathbf{x}+\delta\mathbf{x})$ can be approximated by a quadratic function with a third order error term

$$f(\mathbf{x}+\delta\mathbf{x}) = f(\mathbf{x}) + \mathbf{g}_f^T\,\delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^T\mathbf{H}_f\,\delta\mathbf{x} + O(||\delta\mathbf{x}||^3) \tag{A.394}$$

or even a linear function with a second order error term

$$f(\mathbf{x}+\delta\mathbf{x}) = f(\mathbf{x}) + \mathbf{g}_f^T\,\delta\mathbf{x} + O(||\delta\mathbf{x}||^2) \tag{A.395}$$

- **Multi-variable vector-valued functions:**

  A set of m multi-variable functions $f_1(\mathbf{x}), \cdots, f_m(\mathbf{x})$ can be expressed as a vector function

$$\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), \cdots, f_m(\mathbf{x})]^T \tag{A.396}$$

  The Taylor expansion of the ith component is:

$$f_i(\mathbf{x}+\delta\mathbf{x}) = f_i(\mathbf{x}) + \mathbf{g}_i^T\,\delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^T\mathbf{H}_i\,\delta\mathbf{x} + O(||\delta\mathbf{x}||^3) \qquad (i = 1, \cdots, m) \tag{A.397}$$

  The first two terms of these m components can be written in vector form:

$$\mathbf{f}(\mathbf{x}+\delta\mathbf{x}) \approx \begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \vdots \\ \delta x_n \end{bmatrix} = \mathbf{f}(\mathbf{x}) + \mathbf{J}_\mathbf{f}(\mathbf{x})\,\delta\mathbf{x} \tag{A.398}$$

  where $\mathbf{J}_\mathbf{f}(\mathbf{x})$ is the *Jacobian matrix* defined over the vector function $\mathbf{f}(\mathbf{x})$:

$$\mathbf{J}_\mathbf{f}(\mathbf{x}) = \frac{d}{d\mathbf{x}}\mathbf{f}(\mathbf{x}) = \left[\frac{\partial \mathbf{f}}{\partial x_1}, \cdots, \frac{\partial \mathbf{f}}{\partial x_n}\right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}_{m\times n} \tag{A.399}$$

  Note that the Jacobian matrix is not necessarily square as m and n may not be the same. The 2nd and higher order terms can no longer expressed in matrix.

Note that the Hessian matrix of a function $f(\mathbf{x})$ can be obtained as the Jacobian matrix of the gradient vector of $f(\mathbf{x})$:

$$\mathbf{g}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

$$\mathbf{J}_\mathbf{g}(\mathbf{x}) = \frac{d}{d\mathbf{x}}\mathbf{g}_f(\mathbf{x}) = \mathbf{H}_\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{A.400}$$

A.8 Miscellaneous

A.8.1 Centering Matrix

The centering matrix is defined as

$$\mathbf{C}_n = \mathbf{I}_n - \frac{1}{n}\mathbf{1}\mathbf{1}^T \tag{A.401}$$

where $\mathbf{1} = [1, \cdots, 1]^T$ and $\mathbf{1}\mathbf{1}^T$ is an $n \times n$ matrix with all components equal to 1. when $n = 1, 2\,3$, we have

$$\mathbf{C}_1 = 0; \qquad \mathbf{C}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}; \tag{A.402}$$

$$\mathbf{C}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \tag{A.403}$$

Premultiplying $\mathbf{C}_n$ to an n-D vector $\mathbf{v} = [v_1, \cdots, v_n]^T$, we get

$$\mathbf{C}_n\mathbf{v} = \mathbf{v} - \frac{1}{n}\left(\mathbf{1}\mathbf{1}^T\right)\mathbf{v} = \mathbf{v} - \frac{1}{n}\mathbf{1}\mathbf{1}^T\mathbf{v} = \mathbf{v} - \frac{1}{n}\mathbf{1}\sum_{i=1}^{n} v_i = \mathbf{v} - \bar{v}\mathbf{1} \tag{A.404}$$

where $\bar{v}$ is the mean of all components of $\mathbf{v}$:

$$\bar{v} = \frac{1}{n}\mathbf{1}^T\mathbf{v} = \frac{1}{n}\sum_{i=1}^{n} v_i \tag{A.405}$$

We see that the mean of $\mathbf{v}$ is removed from the resulting vector $\mathbf{C}_n\mathbf{v}$.

Example A.9 $\mathbf{v} = [3,\ -2,\ 5]^T$, with mean $\bar{v} = (3 - 2 + 5)/3 = 2$,

$$\mathbf{u} = \mathbf{C}_3\mathbf{v} = \frac{1}{3}\begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}\begin{bmatrix} 3 \\ -2 \\ 5 \end{bmatrix} = \begin{bmatrix} 3 \\ -2 \\ 5 \end{bmatrix} - \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ -4 \\ 3 \end{bmatrix} \tag{A.406}$$

The mean of the resulting vector $\mathbf{u}$ is $\bar{u} = 0$.

In particular, consider applying $\mathbf{C}_n$ to the following two particular vectors:

- $\mathbf{v} = \mathbf{1}$

$$\mathbf{C}_n\mathbf{v} = \mathbf{C}_n\mathbf{1} = \left[\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^T\right]\mathbf{1} = \mathbf{1} - \frac{1}{n}\mathbf{1}\mathbf{1}^T\mathbf{1} = \mathbf{1} - \frac{1}{n}\mathbf{1}n = \mathbf{0} \tag{A.407}$$

  Treating $\mathbf{C}_n\mathbf{1} = 0\,\mathbf{1}$ as an eigenequation of $\mathbf{C}_n$, we see that $\lambda = 0$ and $\mathbf{v} = \mathbf{1}$ are respectively the eigenvalue and corresponding eigenvector of $\mathbf{C}_n$.

- $\mathbf{v} = [v_1, \cdots, v_n]$ with a zero mean $\sum_{i=1}^n v_i = \mathbf{1}^T\mathbf{v} = 0$ (with $n-1$ degrees of freedom)

$$\mathbf{C}_n\mathbf{v} = \left[\mathbf{I} - \frac{1}{n}\mathbf{1}\mathbf{1}^T\right]\mathbf{v} = \mathbf{v} - \frac{1}{n}\mathbf{1}\mathbf{1}^T\mathbf{v} = \mathbf{v} \tag{A.408}$$

  As $\mathbf{C}_n\mathbf{v} = 1\,\mathbf{v}$, we see that $\lambda = 1$ and any $\mathbf{v}$ satisfying $\mathbf{1}^T\mathbf{v} = 0$ are respectively the eigenvalue of multiplicity $n-1$ and the corresponding eigenvector of $\mathbf{C}_n$.

The centering matrix $\mathbf{C}_n$ has the following properties:

- Idempotence: $\mathbf{C}_n^k = \mathbf{C}_n$

$$\mathbf{C}_n^2 = \mathbf{C}_n\mathbf{C}_n = \left(\mathbf{I}_n - \frac{1}{n}\mathbf{1}\mathbf{1}^T\right)\left(\mathbf{I}_n - \frac{1}{n}\mathbf{1}\mathbf{1}^T\right) = \mathbf{I}_n - \frac{2}{n}\mathbf{1}\mathbf{1}^T + \frac{n}{n^2}\mathbf{1}\mathbf{1}^T = \mathbf{C}_n \tag{A.409}$$

  Once the mean of vector $\mathbf{v}$ is removed, its mean is zero and subsequent removal of mean has not effect.
- $\mathbf{C}_n$ has eigenvalue $\lambda = 1$ of multiplicity $n-1$ and eigenvalue $\lambda = 0$ of multiplicity 1.
- $\mathbf{C}_n$ is singular, its inverse does not exist. Once the mean of vector $\mathbf{v}$ is removed, it cannot be reconstructed by an inverse process.

A.8.2 Normal Direction of a Plane

Consider the following in a d-dimensional space.

- A point is represented as a column vector $\mathbf{x} = [x_1, \cdots, x_d]^T$ containing its d coordinates;
- A surface (or hypersurface if $d > 3$) is represented by an equation $f(x_1, \cdots, x_d) = f(\mathbf{x}) = 0$. Alternatively, by solving the equation above for one of the variables, we get $x_d = g(x_1, \cdots, x_{d-1})$, representing the height along the direction of x_d of a surface defined over a $d-1$ dimensional space.
- A plane (or hyperplane if $d > 3$) is a special surface described by a linear equation:

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = \sum_{i=1}^{d} w_i x_i + b = 0 \tag{A.410}$$

Without loss of generality, we can assume $\mathbf{w}$ is normalized with length $||\mathbf{w}|| = \sqrt{\mathbf{w}^T\mathbf{w}} = 1$ (as otherwise we can divide the equation above by $||\mathbf{w}||$).

- The d-dimensional space is divided by the plane into two regions:

$$R_+ = \{\mathbf{x} \mid \mathbf{w}^T\mathbf{x} + b > 0\}, \quad \text{and} \quad R_- = \{\mathbf{x} \mid \mathbf{w}^T\mathbf{x} + b < 0\} \tag{A.411}$$

- The *normal direction* of a plane $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$ is $\mathbf{w}$, i.e., the projection of any point $\mathbf{x}$ on the plane onto normal direction $\mathbf{w}$ is constant:

$$\mathbf{p}_\mathbf{w}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = -b \tag{A.412}$$

- Solving the linear equation $\mathbf{w}^\mathbf{x} + b = 0$ for any of the d variables, e.g., x_d, we get an alternative representation for the plane, as a function defined over the d-1 dimensional space spanned by the remaining $d-1$ variables $\{x_1, \cdots, x_{d-1}\}$:

$$x_d = g(x_1, \cdots, x_{d-1}) = -\frac{1}{w_d}\left(\sum_{i=1}^{d-1} w_i x_i + b\right) \tag{A.413}$$

  In particular, when $x_1 = \cdots = x_{d-1} = 0$, we get $x_d = -b/w_d$, the intercept of the plane on the axis of x_d.
- When the plane is represented in the form of $x_d = g(x_1, \cdots, x_{d-1}) = -(\sum_{i=1}^{d-1} w_i x_i + b)/w_d$, the normal direction is represented the following, a scaled version of $\mathbf{w}$:

$$\left[-\frac{w_1}{w_d}, \cdots, -\frac{w_{d-1}}{w_d}, -1\right]^T = -\frac{1}{w_d}[w_1, \cdots, w_d]^T = -\frac{1}{w_d}\mathbf{w} \tag{A.414}$$

For example, the following equation defines a plane in 3-D space spanned by x, y and z:

$$f(x, y, z) = ax + by + cz + d = x + 2y + 3z + 4 = 0$$

where $a = 1$, $b = 2$, $c = 3$ and $d = 4$. Solving for z we get

$$z = g(x, y) = -\frac{1}{c}(ax + by + d) = -\frac{1}{3}(x + 2y + 4)$$

When $x = y = 0$, we get $z = -c/d = -4/3$, the intercept of the plane on z-axis. The normal direction of the plane is composed of the coefficients of the three variable x, y and z in $f(x, y, z)$:

$$\mathbf{n} = [a,\ b,\ c]^T = [1,\ 2,\ 3]^T \tag{A.415}$$

or the coefficients of the two variables x and y of $g(x, y)$, with the last component -1:

$$\mathbf{n} = \left[-\frac{a}{c}, -\frac{b}{c}, -1\right]^T = -\frac{1}{c}[a,\ b,\ c]^T = -\frac{1}{3}[1,\ 2,\ 3] \tag{A.416}$$

A.8.3 Convexity

We first consider some concepts regarding convexity:

- *Convex set* or *convex region*: a region in an N-D space in which every point on the straight line between any pair of two points in the region is also inside the region.
- *Convex hull* or *convex envelope, convex closure* of a shape: the smallest convex set containing the shape.
- *Convex function*: a real-valued function $f(\mathbf{x})$ with the property that its *epigraph* or *supergraph*, the region above its graph, is a convex set.
- *Concave function*: if $f(\mathbf{x})$ is convex, then $-f(\mathbf{x})$ is concave.

A multivariate function $f(\mathbf{x})$ is convex if and only if

$$f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2) \tag{A.417}$$

Equivalently, $f(\mathbf{x})$ is concave if and only if

$$f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \geq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2) \tag{A.418}$$

The convex function is *strict* if equality in the relationship never holds. Geometrically, the line segment connecting two points $(\mathbf{x}_1, f(\mathbf{x}_1))$ and $(\mathbf{x}_2, f(\mathbf{x_2}))$ is above the graph of the convex function $f(\mathbf{x})$.

Convexity plays an important role in optimization problems, due to its nice property that if a function $f(\mathbf{x})$ is differentiable and strictly convex, then any point $\mathbf{x}^*$ with a zero gradient $\mathbf{g}_f(\mathbf{x}^*) = \mathbf{0}$ is the unique global minimum of the function. Minimizing a convex objective function, or equivalently maximizing a concave objective function, over a convex domain is called *convex optimization*.

Here are some examples and properties of convex functions:

- Single-variable functions: x^2, $|x|$, e^{ax}, $-\log(x)$
- Any norm of a function or vector is convex. The intersection of the epigraphs is convex.
- The intersection of two convex sets is convex. Also,
- Affine function: $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ is both convex and concave but not strict, as both Eq. (A.417) and Eq. (A.418) become equality when applied to the function.
- If function $f(\mathbf{x})$ is convex, then $a\, f(\mathbf{x}) + b$ is also convex, if $a \geq 0$ so that the convexity is preserved.
- If two functions $f(\mathbf{x})$ and $g(\mathbf{x})$ are both convex, then their linear combination is also convex if the coefficients are no negative:

$$h(\mathbf{x}) = af(\mathbf{x}) + bg(\mathbf{x}), \qquad a, b \geq 0 \tag{A.419}$$

  This can be proven by applying the definition directly to $h(\mathbf{x})$.

- A linear combination of n convex function is also convex if all coefficients are nonnegative:

$$f(\mathbf{x}) = \sum_{i=1}^{n} w_i f_i(\mathbf{x}), \qquad w_1, \cdots, w_n \geq 0 \tag{A.420}$$

- Quadratic functions: $f(\mathbf{x}) = \mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{w}^T\mathbf{x} + b$ is
  - convex if $\mathbf{Q} \geq 0$ is positive semidefinite;
  - strictly convex if $\mathbf{Q} > 0$ is positive definite;
  - concave if $\mathbf{Q} \leq 0$ is negative semidefinite;
  - strictly concave if $\mathbf{Q} < 0$ is negative definite;

If the Hessian matrix $\mathbf{H}_f(\mathbf{x})$, the second order derivative, of $f(\mathbf{x})$ exists and is positive semidefinite, i.e., $\mathbf{H}_f(\mathbf{x}) \geq 0$ for its entire domain, then $f(\mathbf{x})$ is convex. As simple examples in 1-D space, $f_1(x) = x^2$ and $f_2(x) = e^x$ are both convex, and their Hessian, the second order derivatives, $f_1''(x) = 2$ and $f_2''(x) = e^x$ are both greater than zero.

Specifically, let $\mathbf{x}_1$ and $\mathbf{x}_2$ be any two points inside the domain of $f(\mathbf{x})$. Then $f(\mathbf{x})$ is convex if for any $c \in [0,\ 1]$ the following holds:

$$f(c\mathbf{x}_1 + (1-c)\mathbf{x}_2) \leq c\, f(\mathbf{x}_1) + (1-c) f(\mathbf{x}_2) \tag{A.421}$$

where $c\mathbf{x}_1 + (1-c)\mathbf{x}_2$ is any point along the line segment between $\mathbf{x}_1$ and $\mathbf{x}_2$, at which the function value is less than the linear interpolation of the two function values $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$. If the equality does not hold, then $f(\mathbf{x})$ is *strictly convex*. We can also define a *concave function* as the negative of a convex function. If $f(\mathbf{x})$ is convex, then $g(x) = -f(\mathbf{x})$ is *concave*, for which the following inequality holds:

$$g(c\mathbf{x}_1 + (1-c)\mathbf{x}_2) \geq c\, g(\mathbf{x}_1) + (1-c) g(\mathbf{x}_2) \tag{A.422}$$

A convex function $f(\mathbf{x})$ has the property that any local minimum is also a global minimum. Moreover, if $f(\mathbf{x})$ is strictly convex, then it has a single global minimum.

A.8.4 Jensen Inequality

For a convex function $g(\mathbf{x})$, the following holds:

$$g\left(\sum_{i=1}^{n} \lambda_i \mathbf{x}_i\right) \leq \sum_{i=1}^{n} \lambda_i f(\mathbf{x}_i) \tag{A.423}$$

For a concave function $g(\mathbf{x})$, the following holds:

$$g\left(\sum_{i=1}^{n} \lambda_i \mathbf{x}_i\right) \geq \sum_{i=1}^{n} \lambda_i g(\mathbf{x}_i) \tag{A.424}$$

Proof:

Let λ_1 and λ_2 be two arbitrary nonnegative real numbers satisfying $\lambda_1+\lambda_2 = 1$. Then by the definition of convexity, we have

$$f(\lambda_1\mathbf{x}_1 + \lambda_2\mathbf{x}_2) \leq \lambda_1 f(\mathbf{x}_1) + \lambda_2 f(\mathbf{x}_2) \tag{A.425}$$

This results of two points can be generalized to n points by induction. We assume the inequality holds for k points:

$$f\left(\sum_{i=1}^{k} \lambda_i \mathbf{x}_i\right) \leq \sum_{i=1}^{k} \lambda_i f(\mathbf{x}_i) \tag{A.426}$$

Now consider the case of $k+1$ terms with coefficients satisfying $\lambda_1+\cdots+\lambda_{k+1} = 1$:

$$\begin{aligned} f\left(\sum_{i=1}^{k+1} \lambda_i \mathbf{x}_i\right) &= f\left(\lambda_1\mathbf{x}_1 + (1-\lambda_1)\sum_{i=2}^{k+1} \frac{\lambda_i}{1-\lambda_1}\mathbf{x}_i\right) \\ &\leq \lambda_1 f(\mathbf{x}_1) + (1-\lambda_1) f\left(\sum_{i=2}^{k+1} \frac{\lambda_i}{1-\lambda_1}\mathbf{x}_i\right) \end{aligned} \tag{A.427}$$

But as

$$\sum_{i=2}^{k+1} \frac{\lambda_i}{1-\lambda_1} = 1 \tag{A.428}$$

we have the following due to the assumption above:

$$f\left(\sum_{i=2}^{k+1} \frac{\lambda_i}{1-\lambda_1}\mathbf{x}_i\right) \leq \sum_{i=2}^{k+1} \frac{\lambda_i}{1-\lambda_1} f(\mathbf{x}_i) \tag{A.429}$$

Substituting this back we get the desired inequality:

$$f\left(\sum_{i=1}^{k+1} \lambda_i \mathbf{x}_i\right) \leq \lambda_1 f(\mathbf{x}_1) + (1-\lambda_1)\left(\sum_{i=2}^{k+1} \frac{\lambda_i}{1-\lambda_1} f(\mathbf{x}_i)\right) = \sum_{i=1}^{k+1} \lambda_i f(\mathbf{x}_i) \tag{A.430}$$

If $\mathbf{x}$ is a random vector that takes any of the n values $\mathbf{x}_i$ with probability P_i satisfying $P_1 + \cdots + P_n = 1$, then we have

$$f(E(\mathbf{x})) = f\left(\sum_{i=1}^{k} \mathbf{x}_i P_i\right) \leq \sum_{i=1}^{k} f(\mathbf{x}_i) P_i = E(f(\mathbf{x})) \tag{A.431}$$

This result can be generalized to the case where $\mathbf{x}$ is a random vector taking continuous values with $E[\mathbf{x}] = \int \mathbf{x}\, p(\mathbf{x}) d\mathbf{x}$.

A.8.5 Sensitivity and Conditioning

An important concern in numerical analysis is how sensitive the result produced by an algorithm is with respect to the noise in the input data (e.g., observational error) and computational errors (e.g., truncation error). To address this issue, we model a generic algorithm treated as a function with both input and output most generally defined (e.g., real or complex, scalar, vector, matrix, function, etc.), and describe its sensitivity to noise as below:

- A function is *well conditioned (well behaved)* if a small change in the input causes a small change in the output.
- A function is *ill conditioned* (ill behaved) if a small change in the input causes a large change in the output.

How well or ill a function is conditioned can be measured quantitatively by the absolute or relative *condition numbers* defined below:

- The absolute condition number is the upper bound of the ratio between the change in output and change in input:

$$\hat{\kappa} \geq \frac{||\text{change in output}||}{||\text{change in input}||} \tag{A.432}$$

- The *relative condition number* is the upper bound of the ratio between the relative change in output and relative change in input:

$$\kappa \geq \frac{||\text{change in output}||/||\text{output}||}{||\text{change in input}||/||\text{input}||} \tag{A.433}$$

where $||x||$ denotes the *norm* of x. As the relative condition number compares the normalized changes in both input and output, its value is invariant to the specific units used to measure them. Obviously the smaller the condition number, the better the function is conditioned.

Consider specifically the condition numbers of some different systems.

System with Scalar input and output:

We first consider evaluating a scalar function $y = f(x)$ given x. The condition number is:

$$\hat{\kappa} = \lim_{\epsilon \to 0} \sup_{|\delta x| \leq \epsilon} \frac{|\delta f(x)|}{|\delta x|}, \qquad \kappa = \lim_{\epsilon \to 0} \sup_{|\delta x| \leq \epsilon} \frac{|\delta f(x)|/|f(x)|}{|\delta x|/|x|}, \tag{A.434}$$

Here sup is the symbol for *supremum* (least upper bound) and $|x|$ is either the absolute value of x if it is real or the modulus of x if it is complex. Note that in general the condition number is a function of x.

When δx is small, $f(x + \delta x)$ can be approximated by the first two terms of its Taylor series:

$$f(x + \delta x) = f(x) + f'(x)\delta x + f''(x)\frac{\delta x^2}{2} + \cdots \approx f(x) + f'(x)\delta x \tag{A.435}$$

then

$$\delta y = f(x + \delta x) - f(x) \approx f'(x)\delta x \tag{A.436}$$

Taking the absolute value or modulus on both sides, we get

$$\left|\delta y\right| \approx \left|f'(x)\delta x\right| \leq \left|f'(x)\right| \left|\delta x\right| \tag{A.437}$$

and the absolute condition number is:

$$\hat{\kappa} \approx |\delta y|/|\delta x| \approx |f'(x)| \tag{A.438}$$

Dividing both sides by $|y| = |f(x)|$, we further get

$$\frac{|\delta y|}{|y|} \approx \frac{|f'(x)|\,|\delta x|}{|f(x)|} = \frac{|x|\,|f'(x)|}{|f(x)|}\,\frac{|\delta x|}{|x|} \tag{A.439}$$

and the relative condition number:

$$\kappa = \frac{|\delta y|/|y|}{|\delta x|/|x|} \approx \frac{|x|\,|f'(x)|}{|f(x)|} = \frac{|f'(x)|}{|f(x)|/|x|} \tag{A.440}$$

We next consider solving the equation $f(x) = 0$ to find x treated as the output that satisfies $y = f(x) = 0$ treated as the input. The problem can be converted into the evaluation of the function $x = g(y) = f^{-1}(y)$ at $y = 0$. The absolute condition number of this function $g(y) = f^{-1}(y)$ is:

$$\hat{\kappa} = |g'(y)| = \left|\frac{d}{dy}[f^{-1}(y)]\right| = \frac{1}{|f'(x)|} \tag{A.441}$$

which is the reciprocal of the absolute condition number $|f'(x)|$ of the previous evaluating problem, evaluated at the root x^* satisfying $f(x^*) = 0$. We see that the larger $|f'(x)|$, or the smaller $g'(y) = 1/|f'(x)|$, the better conditioned the problem is.

In the figure below shows two functions $|f_1'(x)| > |f_2'(x)|$ for all x. For evaluating function $y = f(x)$, $f_1(x)$ is more ill-conditioned than $f_2(x)$, but for solving $f(x) = 0$, $f_1(x)$ is more well-conditioned than $f_2(x)$.

Consider as an example the following two problems both associated with a linear function $y = f(x) = ax + b$:

- Evaluation of y given x:

  The condition number is $\kappa = \delta y/\delta x$. If x is changed to $x + \delta x$ due to noise, then y becomes $ax + b + a\delta x = y + \delta y$, where $\delta y = a\delta x$, i.e., $\kappa = a$. The larger a, the more ill-conditioned the system is.
- Solving $y = f(x) = 0$ for x given $y = 0$:

  The condition number is $\kappa = \delta x/\delta y$. The given equation can be solved for x:

  $$x = g(y) = \frac{1}{a}(y - b) \tag{A.442}$$

  If y is changed to $y+\delta y$ due to noise, then x becomes $(y-b)/a+\delta y/a = x+\delta x$, where $\delta x = \delta y/a$, i.e., $\kappa = 1/a$. The smaller a, the more ill-conditioned the system is.

Systems with vector input and output

The discussions above can be generalized to a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ with vector input $\mathbf{x} = [x_1, \cdots, x_N]^T$ and vector output $\mathbf{y} = [y_1, \cdots, y_M]^T$. Again consider approximating the function by the first two terms in the Taylor series:

$$\mathbf{f}(\mathbf{x} + \delta\mathbf{x}) \approx \mathbf{f}(\mathbf{x}) + \mathbf{J}_f(\mathbf{x})\delta\mathbf{x} \tag{A.443}$$

where $\mathbf{J}_f(\mathbf{x})$ is the *Jacobian matrix* of the function. We get

$$||\delta\mathbf{y}|| = ||\mathbf{f}(\mathbf{x}+\delta\mathbf{x}) - \mathbf{f}(\mathbf{x})|| \approx ||\mathbf{J}_f(\mathbf{x})\delta\mathbf{x}|| \leq ||\mathbf{J}_f(\mathbf{x})||\ ||\delta\mathbf{x}|| \tag{A.444}$$

and the absolute condition number:

$$\hat{\kappa} = \frac{||\delta\mathbf{y}||}{||\delta\mathbf{x}||} \approx ||\mathbf{J}_f(\mathbf{x})|| \tag{A.445}$$

Dividing both sides of the equation above by $||\mathbf{y}|| = ||\mathbf{f}(\mathbf{x})||$ we further get

$$\frac{||\delta\mathbf{y}||}{||\mathbf{y}||} \approx \frac{||\mathbf{J}_f(\mathbf{x})\delta\mathbf{x}||}{||\mathbf{f}(\mathbf{x})||} \leq \frac{||\mathbf{x}||\ ||\mathbf{J}_f(\mathbf{x})||}{||\mathbf{f}(\mathbf{x})||}\ \frac{||\delta\mathbf{x}||}{||\mathbf{x}||} \tag{A.446}$$

and we can find the relative condition number as

$$\kappa = \frac{||\delta\mathbf{y}||/||\mathbf{y}||}{||\delta\mathbf{x}||/||\mathbf{x}||} \approx \frac{||\mathbf{x}||\ ||\mathbf{J}_f(\mathbf{x})||}{||\mathbf{f}(\mathbf{x})||} \tag{A.447}$$

Consider as an example the following two problems both associated with a linear function:

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} \tag{A.448}$$

- Evaluating $\mathbf{y} = \mathbf{A}\mathbf{x}$ given $\mathbf{x}$:

  The Taylor series of this linear function is

  $$\mathbf{f}(\mathbf{x}+\delta\mathbf{x}) = \mathbf{A}(\mathbf{x}+\delta\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{J}_f\,\delta\mathbf{x} = \mathbf{A}\mathbf{x} + \delta\mathbf{A}\mathbf{x} \tag{A.449}$$

  i.e., $\delta\mathbf{y} = \mathbf{A}\delta\mathbf{x}$ and $||\delta\mathbf{y}|| \leq ||\mathbf{A}||\ ||\delta\mathbf{x}||$. We get the upper bound of the absolute condition number:

  $$\hat{\kappa}(\mathbf{A}) = \frac{||\delta\mathbf{y}||}{||\delta\mathbf{x}||} \leq ||\mathbf{A}|| \tag{A.450}$$

  The relative condition number of this linear system is

  $$\kappa(\mathbf{A}) = \frac{||\mathbf{x}||\ ||\mathbf{J}_f(\mathbf{x})||}{||\mathbf{f}(\mathbf{x})||} = ||\mathbf{A}||\frac{||\mathbf{x}||}{||\mathbf{A}\mathbf{x}||} \tag{A.451}$$

  We also have

  $$||\mathbf{x}|| = ||\mathbf{A}^{-1}\mathbf{A}\mathbf{x}|| \leq ||\mathbf{A}^{-1}||\ ||\mathbf{A}\mathbf{x}||, \quad \text{i.e.,} \quad ||\mathbf{A}\mathbf{x}|| \geq \frac{||\mathbf{x}||}{||\mathbf{A}^{-1}||} \tag{A.452}$$

  Substituting this into the expression of $\kappa(\mathbf{A})$ above we get its upper bound:

  $$\kappa(\mathbf{A}) = ||\mathbf{A}||\frac{||\mathbf{x}||}{||\mathbf{A}\mathbf{x}||} \leq ||\mathbf{A}||\ ||\mathbf{A}^{-1}||\frac{||\mathbf{x}||}{||\mathbf{x}||} = ||\mathbf{A}||\ ||\mathbf{A}^{-1}|| \tag{A.453}$$

  Note that κ is no less than 1:

  $$\kappa(\mathbf{A}) = ||\mathbf{A}^{-1}||\ ||\mathbf{A}|| \geq ||\mathbf{A}^{-1}\mathbf{A}|| = ||\mathbf{I}|| = 1 \tag{A.454}$$

- Solving $\mathbf{A}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ as output given $\mathbf{y}$ as input:

  We assume $\mathbf{A}$ is a full-rank square matrix and therefore invertible, so that there exists a unique solution: $\mathbf{x} = \mathbf{g}(\mathbf{y}) = \mathbf{A}^{-1}\mathbf{y}$. The absolute condition number is $\hat{\kappa}(\mathbf{A}) = ||\mathbf{J}_g(\mathbf{x})|| = ||\mathbf{A}^{-1}||$.

To find the relative condition number $\kappa(\mathbf{A})$, consider $\delta\mathbf{x}$ in the output $\mathbf{x}$ caused by $\delta\mathbf{A}$ as well as $\delta\mathbf{y}$:

$$(\mathbf{A}+\delta\mathbf{A})(\mathbf{x}+\delta\mathbf{x}) = \mathbf{A}\mathbf{x}+\mathbf{A}\delta\mathbf{x}+\delta\mathbf{A}\mathbf{x}+\delta\mathbf{A}\delta\mathbf{x} = \mathbf{y}+\delta\mathbf{y} \tag{A.455}$$

Subtracting $\mathbf{A}\mathbf{x} = \mathbf{y}$ from both sides we get

$$\mathbf{A}\delta\mathbf{x}+\delta\mathbf{A}\mathbf{x}+\delta\mathbf{A}\delta\mathbf{x} = \delta\mathbf{y} \tag{A.456}$$

i.e.,

$$\delta\mathbf{x} = \mathbf{A}^{-1}(\delta\mathbf{y}-\delta\mathbf{A}\mathbf{x}-\delta\mathbf{A}\delta\mathbf{x}) \tag{A.457}$$

Taking norm on both sides, we get

$$\begin{aligned} ||\delta\mathbf{x}|| &\le ||\mathbf{A}^{-1}||\ ||\delta\mathbf{y}-(\delta\mathbf{A}\mathbf{x}+\delta\mathbf{A}\delta\mathbf{x})|| \\ &\le ||\mathbf{A}^{-1}||\ (||\delta\mathbf{y}||+||(\delta\mathbf{A}\mathbf{x}+\delta\mathbf{A}\delta\mathbf{x})||) \\ &\le ||\mathbf{A}^{-1}||\ (||\delta\mathbf{y}||+||\delta\mathbf{A}||\ ||\mathbf{x}||+||\delta\mathbf{A}||\ ||\delta\mathbf{x}||) \end{aligned} \tag{A.458}$$

where we have used $||\mathbf{a}-\mathbf{b}|| \le ||\mathbf{a}||+||\mathbf{b}||$, $||\mathbf{a}+\mathbf{b}|| \le ||\mathbf{a}||+||\mathbf{b}||$, and $||\mathbf{A}\mathbf{x}|| \le ||\mathbf{A}||\ ||\mathbf{x}||$.

Subtracting $||\mathbf{A}^{-1}||\ ||\delta\mathbf{A}||\ ||\delta\mathbf{x}||$ from both sides

$$(1-||\mathbf{A}^{-1}||\ ||\delta\mathbf{A}||)\ ||\delta\mathbf{x}|| \le ||\mathbf{A}^{-1}||\ (||\delta\mathbf{y}||+||\delta\mathbf{A}||\ ||\mathbf{x}||) \tag{A.459}$$

and dividing both sides by $(1-||\mathbf{A}^{-1}||\ ||\delta\mathbf{A}||)\ ||\mathbf{y}||$ (assuming not zero), we get

$$\begin{aligned} \frac{||\delta\mathbf{x}||}{||\mathbf{x}||} &\le \frac{||\mathbf{A}^{-1}||}{1-||\mathbf{A}^{-1}||\ ||\delta\mathbf{A}||}\left(\frac{||\delta\mathbf{y}||}{||\mathbf{x}||}+||\delta\mathbf{A}||\right) = \frac{||\mathbf{A}^{-1}||\ ||\mathbf{A}||}{1-||\mathbf{A}^{-1}||\ ||\delta\mathbf{A}||}\left(\frac{||\delta\mathbf{y}||}{||\mathbf{x}||\ ||\mathbf{A}||}+\frac{||\delta\mathbf{A}||}{||\mathbf{A}||}\right) \\ &\le \frac{||\mathbf{A}^{-1}||\ ||\mathbf{A}||}{1-||\mathbf{A}^{-1}||\ ||\delta\mathbf{A}||}\left(\frac{||\delta\mathbf{y}||}{||\mathbf{y}||}+\frac{||\delta\mathbf{A}||}{||\mathbf{A}||}\right) = \frac{\kappa(\mathbf{A})}{1-\kappa(\mathbf{A})||\delta\mathbf{A}||/||\mathbf{A}||}\left(\frac{||\delta\mathbf{y}||}{||\mathbf{y}||}+\frac{||\delta\mathbf{A}||}{||\mathbf{A}||}\right) \end{aligned} \tag{A.460}$$

where we have used $||\mathbf{y}|| = ||\mathbf{A}\mathbf{y}|| \le ||\mathbf{A}||\ ||\mathbf{x}||$, and $\kappa(\mathbf{A}) = ||\mathbf{A}^{-1}||\ ||\mathbf{A}||$.

In particular,

– if $\delta\mathbf{A} = 0$,

$$\frac{||\delta\mathbf{x}||}{||\mathbf{x}||} \le \kappa(\mathbf{A})\frac{||\delta\mathbf{y}||}{||\mathbf{y}||} \tag{A.461}$$

– if $\delta\mathbf{y} = 0$,

$$\frac{||\delta\mathbf{x}||}{||\mathbf{x}||} \le \frac{\kappa(\mathbf{A})}{1-\kappa(\mathbf{A})||\delta\mathbf{A}||/||\mathbf{A}||}\frac{||\delta\mathbf{A}||}{||\mathbf{A}||} \tag{A.462}$$

We see that the relative condition number is $\kappa(\mathbf{A}) = ||\mathbf{A}||\ ||\mathbf{A}^{-1}||$, same for both problems of evaluation $\mathbf{y} = \mathbf{A}\mathbf{x}$ and solving equation $\mathbf{A}\mathbf{x} = \mathbf{y}$.

We further consider the spectral norm defined as the greatest singular value of the matrix $||\mathbf{A}|| = \sigma_{max}$. Then $||\mathbf{A}^{-1}|| = 1/\sigma_{min}$ is the reciprocal of the smallest singular value of $\mathbf{A}$. Now the relative condition number is

$$\kappa(\mathbf{A}) = ||\mathbf{A}||\ ||\mathbf{A}^{-1}|| = \frac{\sigma_{max}}{\sigma_{min}} \tag{A.463}$$

We see that $\kappa\mathbf{A}$ is large if its σ_{max} and σ_{min} are far apart in values, but it is small otherwise. It is therefore a measurement of how close $\mathbf{A}$ is to singularity. If $\mathbf{A}$ is singular, one or more of its singular values are zero, i.e., $\sigma_{min} = 0$, then $\kappa(\mathbf{A}) = \infty$, and the system $\mathbf{Ax} = \mathbf{y}$ is worst conditioned. On the other hand, if $\mathbf{A} = \mathbf{I}$, i.e., all singular values equal to 1, then $\kappa(\mathbf{I}) = 1$, the smallest condition number possible, and the system $\mathbf{Ix} = \mathbf{y}$ is best conditioned.

In Matlab, the function `cond(A,p)` generates the condition number of matrix A based on its p-norm. When $p = 2$, this is the ratio between the greatest and smallest singular values. The following Matlab commands give the same result:

- `cond(A,2)`
- `norm(A,2)*norm(inv(A),2)`
- `s=svd(A)`, `s(1)/s(end)`

We now further consider the worst case scenarios for both evaluating the matrix-vector product $\mathbf{y} = \mathbf{Ax}$ and solving the linear equation system $\mathbf{Ax} = \mathbf{y}$.

Consider the SVD decomposition $\mathbf{A} = \mathbf{U\Sigma V}^*$ and assume the singular values in $\mathbf{\Sigma}$ are sorted in descending order: $\sigma_1 \geq \cdots \geq \sigma_n$. The SVD of the inverse is $\mathbf{A}^{-1} = \mathbf{V\Sigma}^{-1}\mathbf{U}^*$.

- The worst case for evaluating $\mathbf{Ax} = \mathbf{y}$:

  If $\mathbf{x} = \mathbf{v}_n$ is the minimum right singular vector corresponding to the smallest singular value $\sigma_n = \sigma_{min}$. The equation becomes:

$$\mathbf{y} = \mathbf{Ax} = (\mathbf{U\Sigma V}^*)\,\mathbf{v}_n = \mathbf{U\Sigma}\begin{bmatrix}\mathbf{v}_1^* \\ \vdots \\ \mathbf{v}_n^*\end{bmatrix}\mathbf{v}_n = \mathbf{U}\begin{bmatrix}\sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_n\end{bmatrix}\begin{bmatrix}0 \\ \vdots \\ 0 \\ 1\end{bmatrix}$$

$$= [\mathbf{u}_1, \cdots, \mathbf{u}_n]\begin{bmatrix}0 \\ \vdots \\ 0 \\ \sigma_n\end{bmatrix} = \sigma_n\mathbf{u}_n \tag{A.464}$$

  and

$$\mathbf{y} + \delta\mathbf{y} = \mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \sigma_n\mathbf{u}_n + \mathbf{A}\delta\mathbf{x} \tag{A.465}$$

  The absolute error $||\delta\mathbf{y}|| = ||\mathbf{A}\delta\mathbf{x}||$ may be small if $\delta\mathbf{x}$ is small, but the relative error $||\delta\mathbf{y}||/||\mathbf{y}||$ may be large as $||\mathbf{y}|| = |\sigma_n|\,||\mathbf{u}||$ is small when $\sigma_n = \sigma_{min}$ is small (e.g., close to zero).

- The worst case for solving $\mathbf{y} = \mathbf{Ay}$:

  If $\mathbf{y} = \mathbf{u}_1$ is the maximum left singular vector corresponding to the

largest singular value $\sigma_1 = \sigma_{max}$. The equation becomes:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^*\mathbf{u}_1 = \mathbf{V}\mathbf{\Sigma}^{-1}\begin{bmatrix} \mathbf{u}_1^* \\ \vdots \\ \mathbf{u}_n^* \end{bmatrix}\mathbf{u}_1 = \mathbf{V}\begin{bmatrix} 1/\sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1/\sigma_n \end{bmatrix}\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$= [\mathbf{v}_1, \cdots, \mathbf{v}_n]\begin{bmatrix} 1/\sigma_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \frac{1}{\sigma_1}\mathbf{v}_1 \tag{A.466}$$

and

$$\mathbf{x} + \delta\mathbf{x} = \mathbf{A}^{-1}(\mathbf{y} + \delta\mathbf{y}) = \frac{1}{\sigma_1}\mathbf{v}_1 + \mathbf{A}^{-1}\delta\mathbf{y} \tag{A.467}$$

The absolute error $||\delta\mathbf{x}|| = ||\mathbf{A}^{-1}\delta\mathbf{x}||$ may be small if $\delta\mathbf{y}$ is small, but the relative error $||\delta\mathbf{x}||/||\mathbf{x}||$ may be large as $||\mathbf{x}|| = ||\mathbf{u}||/|\sigma_1|$ is small when $\sigma_1 = \sigma_{max}$ is large.

Example A.10

$$\mathbf{A} = \begin{bmatrix} 5.4321 & 1.2345 \\ 10.8643 & 2.4689 \end{bmatrix}, \tag{A.468}$$

The singular values are $\sigma_1 = 12.45633$, $\sigma_2 = 0.00005$, and

$$||\mathbf{A}|| = \sigma_1 = 12.45633, \qquad ||\mathbf{A}^{-1}|| = 1/\sigma_1 = 18684.68$$
$$\kappa(\mathbf{A}) = ||\mathbf{A}||\,||\mathbf{A}^{-1}|| = 12.45633 \times 18684.68 = 232742.6$$

As the condition number is large, indicating $\mathbf{A}$ is near singular, and $\mathbf{A}\mathbf{x} = \mathbf{y}$ is an ill-conditioned problem. To see this, consider how $\delta\mathbf{y}$ is affected by $\delta\mathbf{x}$ while assuming $\delta\mathbf{A} = 0$:

$$\mathbf{y} + \delta\mathbf{y} = \mathbf{A}^{-1}(\mathbf{x} + \delta\mathbf{x}), \qquad \delta\mathbf{y} = \mathbf{A}^{-1}\delta\mathbf{x} \tag{A.469}$$

The change in output $\mathbf{y}$ is

$$||\delta\mathbf{y}|| \leq ||\mathbf{A}^{-1}||\,||\delta\mathbf{x}|| = 18684.68\,||\delta\mathbf{x}|| \tag{A.470}$$

which is 18684.68 times greater than any change $\delta\mathbf{x}$ in the input. However, for an evaluation problem to find $\mathbf{y} = \mathbf{A}\mathbf{x}$ given $\mathbf{x}$,

$$\mathbf{y} + \delta\mathbf{y} = \mathbf{A}(\mathbf{x} + \delta\mathbf{x}), \qquad \delta\mathbf{y} = \mathbf{A}\delta\mathbf{x} \tag{A.471}$$

The change in output $\mathbf{y}$ is

$$||\delta\mathbf{y} \leq ||\mathbf{A}||\,||\delta\mathbf{x}|| = 12.45633\,||\delta\mathbf{x}|| \tag{A.472}$$

which is not too much greater than the change $\delta\mathbf{x}$ in the input.

Example A.11 Consider solving the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ with

$$\mathbf{A} = \frac{1}{2}\begin{bmatrix} 3.0 & 2.0 \\ 1.0 & 4.0 \end{bmatrix}, \qquad \mathbf{A}^{-1} = \begin{bmatrix} 0.8 & -0.4 \\ -0.2 & 0.6 \end{bmatrix} \tag{A.473}$$

The singular values of $\mathbf{A}$ are $\sigma_1 = 2.5583$ and $\sigma_2 = 0.9772$, the condition number is

$$\kappa = ||\mathbf{A}||\ ||\mathbf{A}^{-1}|| = \frac{\sigma_1}{\sigma_2} = 2.618 \tag{A.474}$$

which is small, indicating this is a well-behaved system. Given two similar inputs $\mathbf{b}_1 = [1,\ 1]^T$ and $\mathbf{b}_2 = [0.99,\ 1.01]^T$ with $\delta\mathbf{b} = [0.01,\ -0.01]^T$, we find the corresponding solutions:

$$\mathbf{x}_1 = \mathbf{A}^{-1}\mathbf{b}_1 = \begin{bmatrix} 0.4//0.4 \end{bmatrix}, \qquad \mathbf{x}_2 = \mathbf{A}^{-1}\mathbf{b}_2 = \begin{bmatrix} 0.388 \\ 0.408 \end{bmatrix}, \qquad \delta\mathbf{x} = \begin{bmatrix} 0.012 \\ -0.008 \end{bmatrix} \tag{A.475}$$

We have

$$||\delta\mathbf{b}|| = 0.0141, \quad ||\mathbf{b}_1|| = 1.4142, \quad ||\delta\mathbf{x}|| = 0.0144, \quad ||\mathbf{x}_1|| = 0.5657 \tag{A.476}$$

and

$$\frac{||\delta\mathbf{x}||/||\mathbf{x}_1||}{||\delta\mathbf{b}||/||\mathbf{b}_1||} = \frac{0.0255}{0.01} = 2.5495 \tag{A.477}$$

Example A.12

$$\mathbf{A} = \frac{1}{2}\begin{bmatrix} 1.000 & 1.000 \\ 1.001 & 0.999 \end{bmatrix}, \qquad \mathbf{A}^{-1} = \begin{bmatrix} -999 & 1000 \\ 1001 & -1000 \end{bmatrix} \tag{A.478}$$

The singular values of $\mathbf{A}$ are $\sigma_1 = 1.0$ and $\sigma_2 = 0.0005$, the condition number is

$$\kappa = ||\mathbf{A}||\ ||\mathbf{A}^{-1}|| = \frac{\sigma_1}{\sigma_2} = 2000 \tag{A.479}$$

indicating matrix $\mathbf{A}$ is close to singularity, and the system is ill-conditioned. The solutions corresponding to the same two inputs $\mathbf{b}_1 = [1,\ 1]^T$ and $\mathbf{b}_2 = [0.99,\ 1.01]^T$ are

$$\mathbf{x}_1 = \mathbf{A}^{-1}\mathbf{b}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad \mathbf{x}_2 = \mathbf{A}^{-1}\mathbf{b}_2 = \begin{bmatrix} 20.99 \\ -19.01 \end{bmatrix}, \qquad \delta\mathbf{x} = \begin{bmatrix} -19.99 \\ 20.01 \end{bmatrix} \tag{A.480}$$

We can further find

$$||\delta\mathbf{b}|| = 0.0141, \quad ||\mathbf{b}_1|| = 1.4142, \quad ||\delta\mathbf{x}|| = 28.2843, \quad ||\mathbf{x}_1|| = 1.4142 \tag{A.481}$$

and the condition number is:

$$\frac{||\delta\mathbf{x}||/||\mathbf{x}_1||}{||\delta\mathbf{b}||/||\mathbf{b}_1||} = \frac{200}{0.01} = 2000 \tag{A.482}$$

We see that a small relative change $||\delta\mathbf{b}||/||\mathbf{b}_1|| = 0.01$ in the input caused a huge change $||\delta\mathbf{x}||/||\mathbf{x}_1|| = 20$ in the output (2000 times greater).

Example A.13 Consider

$$\mathbf{A} = \begin{bmatrix} 3.1 & 2.0 \\ 6.0 & 3.9 \end{bmatrix} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^* = \begin{bmatrix} -0.458 & -0.889 \\ -0.889 & 0.458 \end{bmatrix} \begin{bmatrix} 8.051 & 0.0 \\ 0.0 & 0.011 \end{bmatrix} \begin{bmatrix} -0.839 & -0.544 \\ -0.544 & 0.839 \end{bmatrix} \tag{A.483}$$

with

$$||\mathbf{A}|| = \sigma_1 = 8.05, \quad ||\mathbf{A}^{-1}|| = 1/\sigma_2 = 89.46, \quad \kappa = ||\mathbf{A}||\,||\mathbf{A}^{-1}|| = 720.22 \tag{A.484}$$

As $\mathbf{A}$ is near singular, its condition number κ is large, indicating that the problem associated with this $\mathbf{A}$ is ill-conditioned. We will now specifically consider both problems of evaluating and solving the linear system.

- Solve $\mathbf{A}\mathbf{y} = \mathbf{x}$ for $\mathbf{y}$ given $\mathbf{x}$:

$$\mathbf{y} + \delta\mathbf{y} = \mathbf{A}^{-1}(\mathbf{x} + \delta\mathbf{x}), \quad \delta\mathbf{y} = \mathbf{A}^{-1}\delta\mathbf{x} \tag{A.485}$$

  The absolute condition number is $\hat{\kappa} = ||\mathbf{A}^{-1}|| = 89.46$. The change in output $\mathbf{y}$ is

$$||\delta\mathbf{y}|| \leq ||\mathbf{A}^{-1}||\,||\delta\mathbf{x}|| = \hat{\kappa}\,||\delta\mathbf{x}|| = 89.46\,||\delta\mathbf{x}|| \tag{A.486}$$

  which is $\hat{\kappa} = 89.46$ times greater than any change $\delta\mathbf{x}$ in the input. We further consider the worst case when $\mathbf{x} = \mathbf{u}_{max}$ and the relative error:

$$\mathbf{x} = \mathbf{u}_1 = \begin{bmatrix} -0.458 \\ -0.889 \end{bmatrix}, \quad \mathbf{y} = \mathbf{A}^{-1}\mathbf{x} = \begin{bmatrix} -0.104 \\ -0.068 \end{bmatrix} \tag{A.487}$$

  Even for a small change $\delta\mathbf{x}$ in the input, causing a small change $\delta\mathbf{y}$ in the output:

$$\delta\mathbf{x} = \begin{bmatrix} 0.01 \\ -0.01 \end{bmatrix}, \quad \delta\mathbf{y} = \mathbf{A}^{-1}\delta\mathbf{x} = \begin{bmatrix} 0.656 \\ -1.011 \end{bmatrix} \tag{A.488}$$

  the ratios of the absolute and relative errors are:

$$\frac{||\delta\mathbf{y}||}{||\delta\mathbf{x}||} = \frac{1.205}{0.014} \approx 85.21, \qquad \frac{||\delta\mathbf{y}||/||\mathbf{y}||}{||\delta\mathbf{x}||/||\mathbf{x}||} = \frac{9.702}{0.014} \approx 686 \tag{A.489}$$

- Evaluate $\mathbf{y} = \mathbf{A}\mathbf{x}$ given $\mathbf{x}$:

$$\mathbf{y} + \delta\mathbf{y} = \mathbf{A}(\mathbf{x} + \delta\mathbf{x}), \quad \delta\mathbf{y} = \mathbf{A}\delta\mathbf{x} \tag{A.490}$$

  The absolute condition number is $\hat{\kappa} = ||\mathbf{A}|| = 8.05$. The absolute change in output $\mathbf{y}$ is

$$||\delta\mathbf{y}|| \leq ||\mathbf{A}||\,||\delta\mathbf{x}|| = 8.051\,||\delta\mathbf{x}|| \tag{A.491}$$

  which is not too much greater than the absolute change $\delta\mathbf{x}$ in the input. However, a much greater relative error will result if we consider the worst case when $\mathbf{x} = \mathbf{v}_{min}$:

$$\mathbf{x} = \mathbf{v}_2 = \begin{bmatrix} -0.5444 \\ 0.8388 \end{bmatrix}, \quad \mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} -0.0099 \\ 0.0051 \end{bmatrix} \tag{A.492}$$

Even for a small change $\delta\mathbf{x}$ in the input, causing a small change $\delta\mathbf{y}$ in the output:

$$\delta\mathbf{x} = \begin{bmatrix} 0.01 \\ -0.01 \end{bmatrix}, \quad \delta\mathbf{y} = \mathbf{A}\delta\mathbf{x} = \begin{bmatrix} 0.011 \\ 0.021 \end{bmatrix} \tag{A.493}$$

The ratios of the absolute and relative errors are:

$$\frac{||\delta\mathbf{y}||}{||\delta\mathbf{x}||} \approx 1.68, \qquad \frac{||\delta\mathbf{y}||/||\mathbf{y}||}{||\delta\mathbf{x}||/||\mathbf{x}||} = \frac{2.1207}{0.0141} \approx 150 \tag{A.494}$$

Although the ratio of the absolute errors is small, the ratio of the relative errors is large.

A.8.6 Some Useful Inequalities

- **The interpolation inequality for e^x**

$$e^{ta+(1-t)b} \leq te^a + (1-t)e^b, \qquad t \in [0,1], \qquad a, b \geq 0 \tag{A.495}$$

Proof: Evaluating the exponential function $y = f(x) = e^x$ at two points $x = a$ and $x = b$ we get two points in the x-y plane $(a,\ e^a)$ and $(b,\ e^b)$. The parametric expression of the secant line through these two points is

$$(ta + (1-t)b,\ te^a + (1-t)e^b), \qquad (0 \leq t \leq 1) \tag{A.496}$$

The parametric expression of the function $y = e^x$ between these two points is:

$$(ta + (1-t)b,\ e^{ta+(1-t)b}), \qquad (0 \leq t \leq 1) \tag{A.497}$$

As the $y = e^x$ is convex downward (or concave upward), it is never higher than the secant line function in the interval $a \leq x \leq b$, and we have

$$e^{ta+(1-t)b} \leq te^a + (1-t)e^b \tag{A.498}$$

The equality holds only when $t = 0$ or $t = 1$.

- **Young's inequality**

For any two real positive values $p,\ q > 0$ satisfying

$$\frac{1}{p} + \frac{1}{q} = 1, \qquad q = \frac{p}{p-1}, \quad p = \frac{q}{q-1} \tag{A.499}$$

and any two real numbers α and β, we have

$$\alpha\beta \leq \frac{\alpha^p}{p} + \frac{\beta^q}{q} \tag{A.500}$$

Proof: Substituting $a = p\log\alpha$ and $b = q\log\beta$ into the interpolation inequality for $f(x) = e^x$ above, we get:

$$e^{tp\log\alpha + (1-t)q\log\beta} = e^{tp\log\alpha}e^{(1-t)q\log\beta} = \alpha^{tp}\beta^{(1-t)q}$$
$$\leq te^{p\log\alpha} + (1-t)e^{q\log\beta} = t\alpha^p + (1-t)\beta^q$$

We further let $t = 1/p$, and get $1 - t = 1 - 1/p = 1/q$, the above becomes Young's inequality:

$$\alpha\beta \leq \frac{\alpha^p}{p} + \frac{\beta^q}{q} \tag{A.501}$$

- **Holder's inequality**

$$\left| \sum_i x_i y_i \right| \leq \sum_i |x_i y_i| \leq ||\mathbf{x}||_p ||\mathbf{y}||_q \tag{A.502}$$

Proof: Applying Young's inequality to the following

$$\alpha = \frac{|x_i|}{||\mathbf{x}||_p} = \frac{|x_i|}{(\sum_i |x_i|^p)^{1/p}}, \quad \beta = \frac{|y_i|}{||\mathbf{y}||_q} = \frac{|y_i|}{(\sum_i |y_i|^q)^{1/q}} \tag{A.503}$$

we get

$$\alpha\beta = \frac{|x_i|\,|y_i|}{||\mathbf{x}||_p\,||\mathbf{y}||_q} \leq \frac{\alpha^p}{p} + \frac{\beta^q}{q} = \frac{1}{p}\frac{|x_i|^p}{\sum_i |x_i|^p} + \frac{1}{q}\frac{|y_i|^q}{\sum_i |y_i|^q} \tag{A.504}$$

Taking summation on both sides we get

$$\frac{\sum_i |x_i|\,|y_i|}{||\mathbf{x}||_p\,||\mathbf{y}||_q} \leq \frac{1}{p} + \frac{1}{q} = 1 \tag{A.505}$$

i.e.,

$$\sum_i |x_i y_i| \leq \sum_i |x_i|\,|y_i| \leq ||\mathbf{x}||_p\,||\mathbf{y}||_q \tag{A.506}$$

In particular when $p = 2$, $q = p/(p-1) = 2$, we have

$$|<\mathbf{x}, \mathbf{y}>| = \left| \sum_i x_i \overline{y}_i \right| = \left| \sum_i x_i y_i \right| \leq \sum_i |x_i y_i| \leq ||\mathbf{x}||_2\,||\mathbf{y}||_2 \tag{A.507}$$

This is the Cauchy-Schwarz inequality.

- **Minkowski's inequality**

$$||\mathbf{x} + \mathbf{y}||_p \leq ||\mathbf{x}||_p + ||\mathbf{y}||_p, \qquad p \geq 1 \tag{A.508}$$

Proof:

$$\begin{aligned} ||\mathbf{x} + \mathbf{y}||_p^p &= \sum_i |x_i + y_i|^p = \sum_i |x_i + y_i|\,|x_i + y_i|^{p-1} \leq \sum_i (|x_i| + |y_i|)\,|x_i + y_i|^{p-1} \\ &= \sum_i |x_i|\,|x_i + y_i|^{p-1} + \sum_i |y_i|\,|x_i + y_i|^{p-1} \end{aligned}$$

Applying Holder's inequality to each of the two terms on the right-hand side, we get

$$||\mathbf{x} + \mathbf{y}||_p^p \leq ||\mathbf{x}||_p \left(\sum_i |x_i + y_i|^{(p-1)q} \right)^{1/q} + ||\mathbf{y}||_p \left(\sum_i |x_i + y_i|^{(p-1)q} \right)^{1/q} \tag{A.509}$$

But as $1/p + 1/q = 1$, and

$$q = p/(p-1), \qquad 1/q = (p-1)/p, \qquad (p-1)q = p \tag{A.510}$$

the above becomes:

$$||\mathbf{x}+\mathbf{y}||_p^p \leq ||\mathbf{x}||_p \left(\sum_i |x_i + y_i|^p \right)^{(p-1)/p} + ||\mathbf{y}||_p \left(\sum_i |x_i + y_i|^p \right)^{(p-1)/p} = (||\mathbf{x}||_p + ||\mathbf{y}||_p) \; ||\mathbf{x}+ \tag{A.511}$$

Multiplying both sides by $||\mathbf{x} + \mathbf{y}||_p^{1-p}$, we get Minkowski's inequality.

Appendix B A Review of Probability and Statistics

B.1 Univariate and Multivariate Random Variables

B.1.1 Random Events and Probability Space

- **Random Experiment and Sample Space**

  A *random experiment* is a procedure that can be repeated any number of times and has a set of possible outcomes. The *sample space* S of a certain random experiment is a set of all its possible outcomes, containing either a finite number of discrete (countable) outcomes, or infinite continuous (uncountable) outcomes.

  For example, the random experiment "Randomly pick a card from a deck of 10 cards labeled $0, 1, \cdots, 9$" has a discrete sample space, a set of the 10 possible outcomes: $S = \{0, 1, \cdots, 9\}$.

- **Random Event and Probability Space**

  A *random event* $A \subset S$ is a subset of S, which can be a null or empty set $\emptyset$, a proper subset, e.g., $A = \{0, 2, 4, 6, 8\} \subset S = \{0, 1, \cdots, 9\}$, or the entire S. Event A occurs if the outcome $s \in A$.

  The probability $P(A)$ of an event A is a real-valued function that maps A to a real number $0 \leq P(A) \leq 1$. Specially, $P(\emptyset) = 0$, and $P(S) = 1$.

  For example, the random event "The randomly chosen card has a number smaller than 4" is represented by a subset $A = \{0, 1, 2, 3\} \subset S$, and its probability is $P(A) = 4/10 = 0.4$. Event A occurs if the outcome is one of the member of A (e.g., 2).

  The triple (S, A, P) of sample space S, events A and probability P is called the *probability space*.

- **Intersection, Union, and Complement**

  The *Intersection* $A \cup B$ of events A and B is an event whose outcomes belong to both A and B. If A and B are *mutually exclusive*, i.e., $A \cap B = \emptyset$, then $P(A \cap B) = 0$.

  The *Union* (OR) $A \cup B$ of events A and B is an event whose outcomes belong to either A or B

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{B.1}$$

If A and B are mutually exclusive, then

$$P(A \cup B) = P(A) + P(B) \tag{B.2}$$

The *complement* A^C of an event A is a set of all outcomes not in A. Obviously $A \cup A^C = S$, $A \cap A^C = \emptyset$, and $P(A^C) = 1 - P(A)$.

- **Conditional Probability and Independence**

  The *conditional probability* $P(A|B)$ is the probability of event A given that event B has already occurred, which can be expressed as

$$P(A \cap B) = P(B|A)\,P(A) = P(A|B)\,P(B) \tag{B.3}$$

  which can also expressed in terms of the conditional probability of A given B:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)} \tag{B.4}$$

  Two events A and B are *independent* if and only if

$$P(A \cap B) = P(A)P(B) \tag{B.5}$$

  i.e.,

$$P(A|B) = \frac{P(A,B)}{P(B)} = \frac{P(A)P(B)}{P(B)} = P(A) \tag{B.6}$$

  We further have

$$\left.\begin{array}{l} P(A|B)\,P(B|C) = P(B|A)\,P(A|C) \\ P(A|B,C)\,P(B|C) = P(B|A,C)\,P(A|C) \end{array}\right\} = P(A \cap B|C) \tag{B.7}$$

- **Partition Theorem**

  Let $\{A_1, \cdots, A_n\}$ be a set of *mutually exclusive* events that partition the sample space S, i.e.,

$$\bigcup_{i=1}^{n} A_i = S, \quad \text{and} \quad A_i \cap A_j = \emptyset \quad \forall i \neq j \tag{B.8}$$

  then for any event $B \subset S$, we have

$$P(B) = \sum_{i=1}^{n} P(B \cap A_i) = \sum_{i=1}^{n} P(B|A_i)P(A_i) \tag{B.9}$$

  Proof:

$$\begin{aligned} P(B) = P\,(B \cap S) = P\left(B \cap (\bigcup_{i=1}^{n} A_i)\right) = P\left(\bigcup_{i=1}^{n}(B \cap A_i)\right) \\ = \sum_{i=1}^{n} P(B \cap A_i) = \sum_{i=1}^{n} P(B|A_i)P(A_i) \end{aligned} \tag{B.10}$$

  We further have:

$$P(A_i|B) = \frac{P(B|A_i)\,P(A_i)}{P(B)} = \frac{P(B|A_i)\,P(A_i)}{\sum_{i=1}^{n} P(B|A_i)P(A_i)} \tag{B.11}$$

- **Bayes' Theorem**

  Bayes' theorem is based on the expression for the conditional probability of A given B in Eq. (B.4):

$$p(A|B) = \frac{p(A,B)}{p(B)} = \frac{p(B|A)\,p(A)}{p(B)} \tag{B.12}$$

  If $\{A_1, \cdots, A_n\}$ is a partition of sample space A, then

$$P(B) = \sum_{i=1}^{n} P(B|A_i)\,P(A_i)$$

$$P(A_i|B) = \frac{P(B|A_i)\,P(A_i)}{P(B)} = \frac{P(B|A_i)\,P(A_i)}{\sum_{i=1}^{n} P(B|A_i)\,P(A_i)} \quad (i = 1, \cdots, n) \tag{B.13}$$

  Here the denominator $P(B) = \sum_{i=1}^{n} P(B|A_i)\,P(A_i)$ can be considered as a normalizing factor by which $P(A_i|B)$ is normalized to satisfy $\sum_{i=1}^{n} P(A_i|B) = 1$.

  For example, let $P(A_i)$ be the probability for a student to take any one of some n elective courses $\{A_1, \cdots, A_n\}$, and $P(B|A_i)$ be the probability for the student to get a B grade in course A_i. Given that the student gets a B, the probability for him/her to have taken course A_i is $P(A_i|B)$.

B.1.2 Univariate Random Variables

- **Random Variable**

  A random variable $x = x(s)$ is a real-valued function that maps any outcome $s \in S$ in the sample space into a real number x, which is either discrete if S is countable or continuous if S is uncountable.

- **Cumulative and Density Distribution Functions**

  The *probability mass function (pmf)* of a discrete random variable x with n possible vales $\{x_1, \cdots, x_n\}$ is:

$$P(x = x_i) = P(x_i) = P_i \quad (i = 1, \cdots, n) \tag{B.14}$$

  satisfying

$$0 \le P_i \le 1, \qquad \sum_{i=1}^{n} P_i = 1 \tag{B.15}$$

  The cumulative distribution function is

$$F_x(\xi) = P(x < \xi) = \sum_{x_i < \xi} P(x_i) = \sum_{i=1}^{k} P_i \tag{B.16}$$

  where we assumed $x_1 < \cdots < x_n$ and $\xi = x_{k+1}$.

  The *cumulative distribution function (cdf)* of a continuous random variable x is defined as:

$$F_x(\xi) = P(x < \xi) \tag{B.17}$$

The *probability density function (pdf)* of x is

$$p(x) = \frac{d}{d\xi}F_x(\xi), \quad \text{i.e.} \quad F_x(\xi) = \int_{-\infty}^{\xi} p(x)dx \tag{B.18}$$

and we have

$$P(a \le x < b) = F_x(b) - F_x(a) = \int_a^b p(x)dx \tag{B.19}$$

and

$$F_x(\infty) = \int_{-\infty}^{\infty} p(x)dx = 1 \tag{B.20}$$

A random variable x with pdf $p(x)$ is denoted by $x \sim p(x)$. Sometimes we may also denote a random variable by X and its pdf by $p_X(x)$ for clarity, such as to distinguish the different pdfs of two variables, $X \sim p_X(x)$ and $Y \sim p_Y(y)$.

- **Expectation**

  The *expectation value* (*mean* or *average*) of a random variable x is the average of all possible values it can take weighted by their corresponding probabilities:

$$\mu = \mathrm{E}[x] = \begin{cases} \sum_{i=1}^{n} x_i P_i & x \text{ is discrete} \\ \int_{-\infty}^{\infty} x\, p(x)dx & x \text{ is continuous} \end{cases} \tag{B.21}$$

- **Variance**

  The *variance* of a random variable x is

$$\sigma^2 = \mathrm{Var}[x] = \mathrm{E}[(x-\mu)^2] = \begin{cases} \sum_{i=1}^{n} (x_i - \mu)^2 P_i & x \text{ is discrete} \\ \int_{-\infty}^{\infty} (x-\mu)^2 p(x)dx & x \text{ is continuous} \end{cases} \tag{B.22}$$

  The variance can also be written as:

$$\begin{aligned} \sigma^2 = \mathrm{Var}[x] &= \mathrm{E}[(x-\mu)^2] = \mathrm{E}[x^2 - 2x\mu + \mu^2] \\ &= \mathrm{E}[x^2] - 2\mu_x \mathrm{E}[x] + \mu^2 = \mathrm{E}[x^2] - 2\mu^2 + \mu^2 = \mathrm{E}[x^2] - \mu^2 \end{aligned} \tag{B.23}$$

  The *standard deviation* of x is defined as

$$\sqrt{\mathrm{Var}[x]} = \sqrt{\sigma^2} = \sigma \tag{B.24}$$

  The variance $\sigma_x^2 \ge 0$ is non-negative and it measures the dispersion of the values of variable x, i.e., how widely or narrowly its values distribute over the real axis. Specially if $\sigma_x^2 = 0$, then $x = \mu_x$ is no longer a random variable.

- **Estimation of Mean and Variance**

  When the pmf P_i or pdf $p(x)$ of random variable x is unknown, its mean

μ and variance σ^2 can be estimated based on a set of independent samples $\{x_1, \cdots, x_N\}$ of x:

$$\hat{\mu} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{B.25}$$

$$\hat{\sigma}^2 = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \hat{\mu})^2 \tag{B.26}$$

Note that while estimating σ^2, the sum is divided by $N-1$ instead of N, so that the estimated variance is unbiased, i.e., $\mathrm{E}[\hat{\sigma}^2] = \sigma^2$, as shown below:

$$\begin{aligned}
\mathrm{E}[\hat{\sigma}^2] &= \frac{1}{N-1}\,\mathrm{E}\left[\sum_{n=1}^{N}(x_n - \hat{\mu})^2\right] = \frac{1}{N-1}\,\mathrm{E}\left[\sum_{n=1}^{N}[(x_n - \mu) - (\hat{\mu} - \mu)]^2\right] \\
&= \frac{1}{N-1}\,\mathrm{E}\left[\sum_{n=1}^{N}(x_n - \mu)^2 - 2\sum_{n=1}^{N}(x_n - \mu)(\hat{\mu} - \mu) + \sum_{n=1}^{N}(\hat{\mu} - \mu)^2\right] \\
&\overset{1}{=} \frac{1}{N-1}\,\mathrm{E}\left[\sum_{n=1}^{N}(x_n - \mu)^2\right] - \frac{N}{N-1}\,\mathrm{E}\left[(\hat{\mu} - \mu)^2\right] \\
&\overset{2}{=} \frac{N}{N-1}\sigma^2 - \frac{1}{N-1}\sigma^2 = \sigma^2
\end{aligned} \tag{B.27}$$

where the labeled equal signs are due to the following reasons:
1.The sum in the second term can written as

$$\sum_{n=1}^{N}(x_n - \mu) = N(\hat{\mu} - \mu) \tag{B.28}$$

2.The expectation in the second term is the variance of $\hat{\mu}$ (Eq. (B.81)):

$$\mathrm{E}[(\hat{\mu} - \mu)^2] = \mathrm{Var}[\hat{\mu}] = \sigma^2/N \tag{B.29}$$

- **Standardization**

  A given random variable x with mean μ_x and variance σ_x^2 can be demeaned or centered by removing its mean:

$$y = x - \mu_x \tag{B.30}$$

  so that the resulting variable y has a zero mean

$$\mu_y = \mathrm{E}[y] = \mathrm{E}[x - \mu_x] = \mu_x - \mu_x = 0 \tag{B.31}$$

  but the same variance $\sigma_y^2 = \sigma_x^2$. This conversion is often carried out for mathematical convenience without loss of generality.

  Moreover, variable y obtained above can be further scaled by $1/\sigma_x$ to become

$$z = \frac{y}{\sigma_x} = \frac{x - \mu_x}{\sigma_x} \tag{B.32}$$

so that the resulting variable z not only has a zero mean $\mu_z = 0$ but also a unit variance:

$$\sigma_z^2 = \text{Var}[z] = \text{E}\left[\frac{(x-\mu_x)^2}{\sigma_x^2}\right] = \frac{\text{E}[(x-\mu_x)^2]}{\sigma_x^2} = \frac{\sigma_x^2}{\sigma_x^2} = 1 \tag{B.33}$$

The process combining these two steps to convert x to z is called standardization.

- **The Median and Mode**

  The median $\tilde{x}$ of a random variable x is defined as the middle value of its distribution. If x is discrete, then

$$\tilde{x} = \begin{cases} x_{(n+1)/2} & \text{if } n \text{ is odd} \\ (x_{n/2} + x_{n/2+1})/2 & \text{if } n \text{ is even} \end{cases} \tag{B.34}$$

  If x is continuous, then the median $\tilde{x}$ is defined such that

$$\int_{-\infty}^{\tilde{x}} p(x)dx = \int_{\tilde{x}}^{\infty} p(x)dx = 0.5 \tag{B.35}$$

  The *mode* of variable x is the value that appears most frequently among all possible values:

$$\tilde{x} = \arg\max_x \; p(x), \qquad \text{i.e.,} \quad p(\tilde{x}) = \max_x \; p(x) \tag{B.36}$$

  For example, given $n = 8$ values in $\{1, 2, 3, 3, 4, 4, 4, 5\}$, we can find their mean as

$$\bar{x} = \frac{1}{8}(1+2+3+3+4+4+4+5) = 1\times\frac{1}{8}+2\times\frac{1}{8}+3\times\frac{2}{8}+4\times\frac{3}{8}+5\times\frac{1}{8} = \frac{26}{8} = 3.25 \tag{B.37}$$

  The median and mode are respectively 3.5 and 4.

- **Joint Probability and Marginalization**

  The joint probability of two random variables X and Y is

$$P(x,y) = P((X=x) \cap (Y=y)) = P(x|y)P(y) = P(y|x)P(x) \tag{B.38}$$

  where $P(x|y)$ is the conditional probability of event $X = x$ given that $Y = x$. This definition can be generalized to the joint probability $P(x_1, \cdots, x_n)$ of multiple variables. If these variables are independent, then

$$P(x_1, \cdots, x_n) = \prod_{i=1}^{n} P(x_n) \tag{B.39}$$

  For continuous variables, we have

$$p(x,y) = p(x|y)\, p(y) = p(y|x)\, p(x) \tag{B.40}$$

  If they are independent, then we have

$$p(x|y) = p(x), \quad p(y|x) = p(y), \quad p(x,y) = p(x)\, p(y) \tag{B.41}$$

Given the joint probability $p(x, y)$, we can find the probability $p(x)$ by marginalizing variable y:

$$p(x) = \int p(x, y)\, dy = \int p(x|y)p(y)\, dy = E_y[p(x|y)] \tag{B.42}$$

B.1.3 Multivariate Random Vectors

- **Multiple Random Variables**

  An outcome of a random experiment may need to be described by a set of $n > 1$ random variables $\{x_1, \cdots, x_n\}$, which can be more concisely represented as a *random vector* $\mathbf{x} = [x_1, \cdots, x_n]^T$. For example, in signal processing $\mathbf{x}$ can be a set of time samples $x_i = x(t_i)$ of a random signal $x(t)$ also called a *stochastic process*, and in pattern recognition $\mathbf{x}$ can be a *pattern* representing an object of interest, in terms of its n *features* $\{x_1, \cdots, x_n\}$ that describe the object.

- **Joint Distribution Function and Density Function**

  The *joint distribution function* of a random vector $\mathbf{x}$ is defined as

$$\begin{aligned} F_{\mathbf{x}}(u_1, \cdots, u_n) &= P(x_1 < u_1, \cdots, x_n < u_n) \\ &= \int_{-\infty}^{u_1} \cdots \int_{-\infty}^{u_n} p(\xi_1, \cdots, \xi_n) d\xi_1 \cdots d\xi_n \end{aligned} \tag{B.43}$$

  where $p(\xi_1, \cdots, \xi_n)$ is the *joint density function* of the random vector $\mathbf{x}$ in the n-dimensional vector space. In particular, if all these random variables are independent, then $p(x_1, \cdots, x_n) = p(x_1) \cdots p(x_n) = \prod_{i=1}^{n} p(x_i)$.

- **Mean Vector**

  The *expectation* or *mean* of a random variable x_i is defined as

$$\mu_i = \mathrm{E}[x_i] = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \xi_i\, p(\xi_1, \cdots, \xi_n)\, d\xi_1 \cdots d\xi_n \tag{B.44}$$

  The *mean vector* of a random vector $\mathbf{x}$ is defined as

$$\mathbf{m} = \mathrm{E}[\mathbf{x}] = [\mathrm{E}[x_1], \cdots, \mathrm{E}[x_n]]^T = [\mu_1, \cdots, \mu_n]^T \tag{B.45}$$

  which can be interpreted as the center of gravity of an n-dimensional object with $p(x_1, \cdots, x_n)$ being the density function.

- **Covariance Matrix**

  The *covariance* of any two variables x_i and x_j is defined as

$$\begin{aligned} \sigma_{ij}^2 &= \mathrm{Cov}[x_i, x_j] = \mathrm{E}[(x_i - \mu_i)(x_j - \mu_j)] \\ &= \mathrm{E}[x_i x_j] - \mathrm{E}[x_i]\mu_j - \mu_i\, \mathrm{E}[x_j] + \mu_i \mu_j = \mathrm{E}[x_i x_j] - \mu_i \mu_j \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \xi_i \xi_j\, p(\xi_i, \xi_j)\, d\xi_i\, d\xi_j - \mu_i \mu_j \end{aligned} \tag{B.46}$$

  where $\mathrm{E}[x_i x_j] = r_{ij} = \sigma_{ij}^2 + \mu_i \mu_j$ is the *correlation* of variables x_i and x_j. Note that $\sigma_{ij}^2 = \sigma_{ji}^2$ is either positive or negative, and so is r_{ij}. But the variance $\sigma_{ii}^2 = \mathrm{E}[(x_i - \mu_i)^2] \geq 0$ is nonnegative. It can be interpreted as the

amount of dynamic energy or information contained in x_i as a sample of a signal represented by $\mathbf{x}$.

If $\sigma_{ij}^2 = 0$, then variables x_i and x_j said to be *uncorrelated*. If x_i and x_j are independent, i.e., $p(x_i, x_j) = p(x_i)p(x_j)$, then $\mathrm{E}[x_i x_j] = \mu_i \mu_j$. and the covariance defined above is $\sigma_{ij}^2 = \mathrm{E}[x_i]\ \mathrm{E}[x_j] - \mu_i \mu_j = 0$, i.e., x_i and x_j are uncorrelated. However, two uncorrelated variables are not necessarily independent. If $\mathbf{x}$ is normally distributed, then its components are independent if and only they are uncorrelated.

The *covariance matrix* of a random vector $\mathbf{x}$ is defined as

$$\begin{aligned} \boldsymbol{\Sigma}_x &= \mathrm{E}[(\mathbf{x} - \mathbf{m})\,(\mathbf{x} - \mathbf{m})^T] = \mathrm{E}[\mathbf{x}\mathbf{x}^T] - \mathbf{m}\mathbf{m}^T = \mathbf{R} - \mathbf{m}\mathbf{m}^T \\ &= \begin{bmatrix} \sigma_{11}^2 & \cdots & \sigma_{1n}^2 \\ \vdots & \ddots & \vdots \\ \sigma_{n1}^2 & \cdots & \sigma_{nn}^2 \end{bmatrix} \end{aligned} \tag{B.47}$$

where $\mathbf{R} = \mathrm{E}[\mathbf{x}\mathbf{x}^T] = \boldsymbol{\Sigma}_x + \mathbf{m}\mathbf{m}^T$ is the *correlation matrix*:

$$\mathbf{R}_x = \mathrm{E}[\mathbf{x}\mathbf{x}^T] = \begin{bmatrix} \mathrm{E}[x_1^2] & \cdots & \mathrm{E}[x_1 x_n] \\ \vdots & \ddots & \vdots \\ \mathrm{E}[x_n x_1] & \cdots & \mathrm{E}[x_n^2] \end{bmatrix}, = \begin{bmatrix} r_{11}^2 & \cdots & r_{1n}^2 \\ \vdots & \ddots & \vdots \\ r_{n1}^2 & \cdots & r_{nn}^2 \end{bmatrix} \tag{B.48}$$

If $\mathbf{m} = \mathbf{0}$, then $\boldsymbol{\Sigma} = \mathbf{R}$.

The trace of $\boldsymbol{\Sigma}_x$ as the sum of all n variances of x_i can be interpreted as the total dynamic energy or information contained in $\mathbf{x}$:

$$\mathrm{tr}\ \boldsymbol{\Sigma} = \sum_{i=1}^{n} \sigma_i^2 = \sum_{i=1}^{n} \lambda_i \tag{B.49}$$

where $\lambda_1, \cdots, \lambda_n$ are the real and non-negative eigenvalues of the positive semi-definite matrix $\boldsymbol{\Sigma}$.

Specially if $\sigma_{ij} = 0$ for all $i \neq j$, the random vector $\mathbf{x}$ is uncorrelated, and the covariance matrix becomes diagonal $\boldsymbol{\Sigma} = \mathrm{diag}(\sigma_1^2, \cdots, \sigma_n^2)$ with $\lambda_i = \sigma_i^2$.

- **Correlation Coefficient**

  The *Pearson correlation coefficient* between two random variables x_i and x_j is the normalized covariance σ_{ij}^2, when it is divided by the standard deviations σ_i and σ_j:

$$\rho_{ij} = \frac{\sigma_{ij}^2}{\sqrt{\sigma_i^2 \sigma_j^2}} = \frac{\sigma_{ij}^2}{\sigma_i \sigma_j} \tag{B.50}$$

  As the covariance can be considered as an inner product of x_i and x_j

$$\sigma_{ij}^2 = \mathrm{E}[(x - \mu_i)\,(x - \mu_j)] = \langle x_i,\, x_j \rangle \tag{B.51}$$

  the Cauchy-Schwarz inequality (Eq. (A.23)) applies:

$$|\langle x_i,\ x_j \rangle|^2 = \sigma_{ij}^4 \leq \langle x_i,\, x_i \rangle \langle x_j,\, x_j \rangle = \sigma_i^2 \sigma_j^2 \tag{B.52}$$

i.e.,

$$\left(\frac{\sigma_{ij}^2}{\sigma_i \sigma_j}\right)^2 = \rho^2 \leq 1, \qquad -1 \leq \rho \leq 1 \tag{B.53}$$

The correlation coefficient of two random variables x and y can estimated by the *sample correlation coefficient* based on a set of samples of the two variables $\{(x_i, y_i), \quad i = 1, \cdots, k\}$:

$$\hat{\rho}_{xy} = \frac{\hat{\sigma}_{xy}^2}{\hat{\sigma}_x \hat{\sigma}_y} = \frac{\sum_{i=1}^{k}(x_i - \mu_x)(y_i - \mu_y)}{\sqrt{\left(\sum_{i=1}^{k}(x_i - \mu_x)^2\right)\left(\sum_{i=1}^{k}(y_i - \mu_y)^2\right)}} \tag{B.54}$$

- **Mean and Covariance under Orthogonal Transforms**

  Given an orthogonal matrix $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_n]$ composed of n orthogonal column vectors satisfying $\mathbf{a}_i^T \mathbf{a}_j = \delta_{ij}$, i.e., $\mathbf{A}^T = \mathbf{A}^{-1}$, an orthogonal transform of $\mathbf{x}$ can be defined as

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \mathbf{A}^T \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} \mathbf{x}, \qquad \text{i.e.} \quad y_i = \mathbf{a}_i^T \, \mathbf{x}, \quad (i = 1, \cdots, n) \tag{B.55}$$

  The resulting $\mathbf{y}$ is also a random vector. The inverse transform is:

$$\mathbf{x} = \mathbf{A}\mathbf{y} = [\mathbf{a}_1, \cdots, \mathbf{a}_n] \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} y_i \mathbf{a}_i \tag{B.56}$$

  Consider the squared norm of vector $\mathbf{x}$, the total amount of energy contained in $\mathbf{x}$:

$$||\mathbf{x}||^2 = \mathbf{x}^T \mathbf{x} = \left(\sum_{i=1}^{n} y_i \mathbf{a}_i\right)^T \left(\sum_{j=1}^{n} y_j \mathbf{a}_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \, \mathbf{a}_i^T \mathbf{a}_j = \sum_{i=1}^{n} y_i^2 = ||\mathbf{y}||^2 \tag{B.57}$$

  This is Parseval's theorem, indicating that the total energy is conserved under any orthogonal transform.

  The mean vector $\mathbf{m}_y$ and the covariance matrix $\mathbf{\Sigma}_y$ of $\mathbf{y}$ are related to the $\mathbf{m}_x$ and $\mathbf{\Sigma}_x$ of $\mathbf{x}$ by:

$$\mathbf{m}_y = \mathrm{E}[\mathbf{y}] = \mathrm{E}[\mathbf{A}^T \mathbf{x}] = \mathbf{A}^T \, \mathrm{E}[\mathbf{x}] = \mathbf{A}^T \mathbf{m}_x \tag{B.58}$$

  and

$$\begin{aligned} \mathbf{\Sigma}_y &= \mathrm{E}[\mathbf{y}\mathbf{y}^T] - \mathbf{m}_y \mathbf{m}_y^T = \mathrm{E}[\mathbf{A}^T \mathbf{x}\mathbf{x}^T \mathbf{A}] - \mathbf{A}^T \mathbf{m}_x \mathbf{m}_x^T \mathbf{A} \\ &= \mathbf{A}^T \, \mathrm{E}[\mathbf{x}\mathbf{x}^T]\mathbf{A} - \mathbf{A}^T \mathbf{m}_x \mathbf{m}_x^T \mathbf{A} = \mathbf{A}^T [\mathrm{E}[\mathbf{x}\mathbf{x}^T] - \mathbf{m}_x \mathbf{m}_x^T]\mathbf{A} = \mathbf{A}^T \mathbf{\Sigma}_x \mathbf{A} \end{aligned} \tag{B.59}$$

  The orthogonal transform does not change the trace of $\mathbf{\Sigma}$: Due to the commutativity of trace: $\mathrm{tr}(\mathbf{AB}) = \mathrm{tr}(\mathbf{BA})$, we have:

$$\mathrm{tr}\, \mathbf{\Sigma}_\mathbf{y} = \mathrm{tr}(\mathbf{A}^T \mathbf{\Sigma}_x \mathbf{A}) = \mathrm{tr}(\mathbf{A}^T \mathbf{A} \mathbf{\Sigma}_x) = \mathrm{tr}\, \mathbf{\Sigma}_x \tag{B.60}$$

i.e., the total amount of dynamic energy or information contained in $\mathbf{x}$ is conserved by the orthogonal transform $\mathbf{y} = \mathbf{A}^T\mathbf{x}$.

- **Estimation of m and Σ**

  When $p(\mathbf{x}) = p(x_1, \cdots, x_n)$ is not known, $\mathbf{m}$ and $\boldsymbol{\Sigma}$ cannot be found as defined in Eqs. (B.45) and (B.46). However, they can be estimated if a set of samples $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ of $\mathbf{x}$ is available. The sample mean $\hat{\mathbf{m}}$ is:

$$\hat{\mathbf{m}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \tag{B.61}$$

  and the unbiased *sample covariance* is:

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \hat{\mathbf{m}})(\mathbf{x}_n - \hat{\mathbf{m}})^T \tag{B.62}$$

  If all N samples in the dataset $\mathbf{X}$ are independent, then the rank of the $n \times n$ sample covariance matrix $\hat{\boldsymbol{\Sigma}}$ is at most $n-1$, due to the constraining equation:

$$\sum_{n=1}^{N}(\mathbf{x}_n - \hat{\mathbf{m}}) = \sum_{n=1}^{N}\mathbf{x}_n - N\hat{\mathbf{m}} = \mathbf{0} \tag{B.63}$$

  When $N < n-1$, i.e., the sample size is smaller than the rank of the sample covariance matrix $\hat{\boldsymbol{\Sigma}}$, then it does not have a full rank and is not invertible.

- **Standardization**

  Same as in the case of univariate random variables, a multivariate random vector $\mathbf{x}$ with mean $\mathbf{m}_x$ and covariance $\boldsymbol{\Sigma}_x$ can also be standardized. First, we remove the mean to get $\mathbf{y} = \mathbf{x} - \mathbf{m}_x$, so that $\mathbf{m}_y = \mathbf{0}$ and the same covariance $\boldsymbol{\Sigma}_y = \boldsymbol{\Sigma}_x$. We then consider the eigen-equation of this matrix (Eq. (A.95)):

$$\mathbf{V}^{-1}\boldsymbol{\Sigma}_x\mathbf{V} = \mathbf{V}^T\boldsymbol{\Sigma}_x\mathbf{V} = \boldsymbol{\Lambda} \tag{B.64}$$

  As $\boldsymbol{\Sigma}_x$ is positive semidefinite, its eigenvalues in the diagonal eigenvalue matrix $\boldsymbol{\Lambda}$ are non-negative, and its eigenvector matrix $\mathbf{V}^{-1} = \mathbf{V}^T$ is orthogonal. We can now carry out another transform to get

$$\mathbf{z} = (\boldsymbol{\Lambda}^{-1/2}\mathbf{V}^T)\mathbf{y} \tag{B.65}$$

  with zero mean $\mathbf{m}_z = \mathbf{m}_y = \mathbf{0}$ and covariance

$$\begin{aligned}\boldsymbol{\Sigma}_z &= \mathrm{E}[\mathbf{z}\mathbf{z}^T] = \mathrm{E}\left[(\boldsymbol{\Lambda}^{-1/2}\mathbf{V}^T\mathbf{y})(\boldsymbol{\Lambda}^{-1/2}\mathbf{V}^T\mathbf{y})^T\right] \\ &= \boldsymbol{\Lambda}^{-1/2}\mathbf{V}^T\,\mathrm{E}[\mathbf{y}\mathbf{y}^T]\mathbf{V}\boldsymbol{\Lambda}^{-1/2} = \boldsymbol{\Lambda}^{-1/2}(\mathbf{V}^T\boldsymbol{\Sigma}_x\mathbf{V})\boldsymbol{\Lambda}^{-1/2} \\ &= \boldsymbol{\Lambda}^{-1/2}\boldsymbol{\Lambda}\boldsymbol{\Lambda}^{-1/2} = \mathbf{I}\end{aligned} \tag{B.66}$$

  i.e., all components of $\mathbf{z}$ are decorrelated and they all have zero mean and unit variance.

B.1.4 Function of Random Variables

A function $Y = f(X)$ of a random variable X with pdf $p_X(x)$ is also a random variable, and its pdf $p_Y(y)$ can be obtained based on $p_X(x)$ if the function is differentiable and invertible, i.e., a unique $x = f^{-1}(y)$ exists. We first consider its cdf in the following two cases. First, if $y = f(x)$ is a monotonically increasing function ($dy/dx > 0$), then its cdf is

$$F_Y(y) = P(Y < y) = P(f(X) < y) = P(X < f^{-1}(y)) = \int_{-\infty}^{x} p_X(u)du\Big|_{x=f^{-1}(y)} \tag{B.67}$$

based on which we can find its pdf:

$$p_Y(y) = \frac{d}{dy}F_Y(y) = \frac{d}{dy}\int_{-\infty}^{x} p_X(u)du\Big|_{x=f^{-1}(y)} = \left[p_X(x)\frac{dx}{dy}\right]_{x=f^{-1}(y)} \tag{B.68}$$

Next, if $y = f(x)$ is monotonically decreasing ($dy/dx < 0$), then its cdf is:

$$F_Y(y) = P(Y < y) = P(f(X) < y) = P(X > f^{-1}(y)) = \int_{x}^{\infty} p_X(u)du\Big|_{x=f^{-1}(y)} \tag{B.69}$$

and its pdf is:

$$p_Y(y) = \frac{d}{dy}F_Y(y) = \frac{d}{dy}\int_{x}^{\infty} p_X(u)du\Big|_{x=f^{-1}(y)} = \left[-p_X(x)\frac{dx}{dy}\right]_{x=f^{-1}(y)} \tag{B.70}$$

Combining these two cases, we get

$$p_Y(y) = \left[p_X(x)\left|\frac{dx}{dy}\right|\right]_{x=f^{-1}(y)} \tag{B.71}$$

Alternatively, consider the probability for $a \le x \le b$, same as the probability for $f(a) \le y = f(x) \le f(b)$, i.e.,

$$\int_a^b p_X(x)dx = \int_{f(a)}^{f(b)} p_Y(y)dy \tag{B.72}$$

If we assume the interval $b-a$ is infinitesimally small and so is $f(b)-f(a)$, then $p_X(x)$ and $p_Y(y)$ can be assumed to be constants, we have

$$p_X(x)dx = p_Y(y)dy, \qquad \text{i.e.,} \quad p_Y(y) = \left[p_X(x)\left|\frac{dx}{dy}\right|\right]_{x=f^{-1}(y)} \tag{B.73}$$

where the absolute value $|dx/dy| \ge 0$ guarantees that $p_Y(y) > 0$ as well as $p_X(x) > 0$.

As a simple example, consider the pdf of $Y = f(X) = cX$, where c is the constant and $X \sim p_X(x)$, i.e., $x = f^{-1}(y) = y/c$ and $dx/dy = 1/c$. Now we have

$$Y = cX \sim p_Y(y) = \left[p_X(x)\left|\frac{dx}{dy}\right|\right]_{x=f^{-1}(y)} = \frac{1}{c}\, p_X\left(\frac{y}{c}\right) \tag{B.74}$$

with mean and variance:

$$\begin{aligned}\mu_y &= \mathrm{E}[Y] = \mathrm{E}[cX] = c\ \mathrm{E}[X] = c\,\mu_x \\ \sigma_y^2 &= \mathrm{E}[Y^2 - \mu_y^2] = \mathrm{E}[(cX)^2 - (c\mu_x)^2] = c^2\sigma_x^2\end{aligned} \tag{B.75}$$

We next consider the mean and variance of $y = x_1 + \cdots + x_N$ as the sum of N random variables:

$$\mu_y = \mathrm{E}[y] = \mathrm{E}\left[\sum_{i=1}^{N} x_i\right] = \sum_{i=1}^{n} \mathrm{E}[x_i] = \sum_{i=1}^{N} \mu_{x_i} \tag{B.76}$$

We also have

$$\mathrm{E}[y^2] = \mathrm{E}\left[\left(\sum_{i=1}^{N} x_i\right)^2\right] = \mathrm{E}\left[\sum_{i=1}^{N}\sum_{j=1}^{N} x_i x_j\right] = \sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{E}[x_i x_j] \tag{B.77}$$

and

$$(\mathrm{E}[y])^2 = \left(\sum_{i=1}^{N} \mathrm{E}[x_i]\right)^2 = \sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{E}[x_i]\,\mathrm{E}[x_j] \tag{B.78}$$

then we get

$$\begin{aligned}\sigma_y^2 = \mathrm{Var}(y) &= \mathrm{E}[y^2] - (\mathrm{E}[y])^2 = \sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{E}[x_i x_j] - \sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{E}[x_i]\,\mathrm{E}[x_j] \\ &= \sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{Cov}[x_i,\, x_j] = \sum_{i=1}^{N} \mathrm{Var}[x_i] + 2\sum_{i<j} \mathrm{Cov}[x_i,\, x_j]\end{aligned} \tag{B.79}$$

Specially if all x_i are uncorrelated (or more specially independent), i.e., $\mathrm{Cov}[x_i,\, x_j] = 0$ for all $i \neq j$, we have

$$\sigma_y^2 = \sum_{i=1}^{N} \mathrm{Var}[x_i] = \sum_{i=1}^{N} \sigma_{x_i}^2 \tag{B.80}$$

As one more example, consider the variance of the estimated mean in Eq. (B.25) based on N i.i.d. samples:

$$\mathrm{Var}[\hat{\mu}] = \mathrm{Var}\left[\frac{1}{N}\sum_{n=1}^{N} x_n\right] = \frac{1}{N^2}\sum_{n=1}^{N} \mathrm{Var}[x_n] = \frac{\sigma^2}{N} \tag{B.81}$$

We next consider a random vector $\mathbf{x}$ and a function

$$\mathbf{y} = [y_1, \cdots, y_m]^T = [f_1(\mathbf{x}), \cdots, f_m(\mathbf{x})]^T \mathbf{f}(\mathbf{x}) \tag{B.82}$$

If the function is differentiable and invertible, i.e., $\mathbf{x} = \mathbf{f}^{-1}(\mathbf{y})$ exists, then we can find the pdf of $\mathbf{y}$ given the pdf $\mathbf{x}$:

$$p_Y(\mathbf{y}) = \left[p_X(\mathbf{x})\left|\det \mathbf{J}_{f^{-1}}\right|\right]_{\mathbf{x}=\mathbf{f}^{-1}(\mathbf{y})} \tag{B.83}$$

where $|\det \mathbf{J}_{f^{-1}}|$ is the absolute value of the determinant of the Jacobian matrix:

$$\mathbf{J}_{f^{-1}} = \begin{bmatrix} \frac{\partial x_1}{\partial y_1} & \cdots & \frac{\partial x_1}{\partial y_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_m}{\partial y_1} & \cdots & \frac{\partial x_m}{\partial y_n} \end{bmatrix}_{\mathbf{x}=\mathbf{f}^{-1}(\mathbf{y})} \tag{B.84}$$

Specially, for a linear function $\mathbf{y} = \mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is an invertible square matrix, we have $\mathbf{J}_f = \mathbf{A}$ and $\mathbf{J}_{f^{-1}} = \mathbf{A}^{-1}$, and

$$p_Y(\mathbf{y}) = p_Y(\mathbf{A}\mathbf{x}) = \left[|\det \mathbf{A}^{-1}| p_X(x)\right]_{\mathbf{x}=\mathbf{A}^{-1}\mathbf{y}} \tag{B.85}$$

The mean and covariance of a function of a random vector can be approximated based on its Taylor expansion near the mean. Consider $y = f(x)$ as a nonlinear function of a random variable x. Its Taylor expansion near the mean μ_x is:

$$y = f(x) = f(\mu_x) + f'(\mu_x)(x-\mu_x) + \frac{f''(\mu_x)}{2}(x-\mu_x)^2 + \cdots \approx f(\mu_x) + f'(\mu_x)(x-\mu_x) \tag{B.86}$$

Taking the mean on both side we get

$$\mu_y = \mathrm{E}[y] \approx \mathrm{E}[f(\mu_x)] + f'(\mu_x)\,\mathrm{E}[x-\mu_x] = f(\mu_x) \tag{B.87}$$

Taking the variance on both sides we get

$$\begin{aligned} \mathrm{Var}[y] &\approx \mathrm{Var}[f(\mu_x) + f'(\mu_x)(x-\mu_x)] = \mathrm{Var}[f'(\mu_x)(x-\mu_x)] \\ &= f'^2(\mu_x)\,\mathrm{Var}[x] = f'^2(\mu_x)\sigma_x^2 \end{aligned} \tag{B.88}$$

For a function $y = f(\mathbf{x})$ of a random vector $\mathbf{x}$, its Taylor expansion in the neighborhood of the mean vector $\mathbf{m}_x$ is:

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{m}_x) + \mathbf{g}(\mathbf{m}_\mathbf{x})^T(\mathbf{x}-\mathbf{m}_x) + \frac{1}{2}(\mathbf{x}-\mathbf{m}_x)^T\mathbf{H}(\mathbf{m}_x)(\mathbf{x}-\mathbf{m}_x) + \cdots \\ &\approx f(\mathbf{m}_x) + \mathbf{g}(\mathbf{m}_\mathbf{x})^T(\mathbf{x}-\mathbf{m}_x) \end{aligned} \tag{B.89}$$

where

$$\mathbf{g}(\mathbf{m}_x) = \begin{bmatrix} \frac{\partial f\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f\mathbf{x})}{\partial x_n} \end{bmatrix}_{\mathbf{m}_x}, \quad \mathbf{H}(\mathbf{m}_x) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}_{\mathbf{m}_x} \tag{B.90}$$

are the gradient vector and Hessian matrix of $f(\mathbf{x})$ evaluated at $\mathbf{m}_x$. Taking the mean of the expression above

$$\mu_y \approx Ef(\mathbf{x}) \approx \mathrm{E}\left[f(\mathbf{m}_x)\right]\mathbf{g}\,\mathrm{E}[\mathbf{x}-\mathbf{m}_x] = f(\mathbf{m}_x) \tag{B.91}$$

Taking the variance we get

$$\begin{aligned} \sigma_y^2 &= \mathrm{Var}\left[f(\mathbf{x})\right] \approx \mathrm{Var}\left[f(\mathbf{m}_x) + \mathbf{g}(\mathbf{m}_x)^T(\mathbf{x}-\mathbf{m}_x)\right] = \mathrm{Var}\left[\mathbf{g}(\mathbf{m}_x)^T(\mathbf{x}-\mathbf{m}_x)\right] \\ &= \mathrm{Var}\left[\sum_{i=1}^n \frac{\partial f}{\partial x_i}(x_i - \mu_{x_i})\right] = \sum_{i=1}^n \left(\frac{\partial f}{\partial x_i}\right)^2 \sigma_{x_i}^2 \end{aligned} \tag{B.92}$$

or

$$\sigma_y = \sqrt{\sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i}\right)^2 \sigma_{x_i}^2} \tag{B.93}$$

B.1.5 The Normal Distribution

We first consider the central limit theorem (CLT), which states that the distribution of the sum of a number of independent and identically distributed (i.i.d.) random variables of arbitrary distribution tends to a normal distribution as the number of variables approaches infinity.

Specifically, Let $\{x_1, \cdots, x_N\}$ be a set of i.i.d. samples drawn from an arbitrary distribution with mean μ and variance σ^2. Then the the CLT states that the distribution of the sample average

$$\bar{x}_N = \frac{1}{N}\sum_{n=1}^{N} x_i \tag{B.94}$$

tends to a normal distribution $p(x) = \mathcal{N}(x, \mu, \sigma^2/N)$.

For example, the face value of a dice has a uniform distribution $P(v = 1) = \cdots = P(v = 6) = 1/6$. But the distribution of the sum of a pair of dice is no longer uniform. It has a maximum probability at the mean of 7. As the number of dice increases, the distribution of the sum of the face values approaches a Gaussian, as illustrated in the figure below showing the estimated distribution of the sum of $N = 1, 2, 4$ and 8 dice.

The normal or Gaussian distribution is the most important pdf widely used to model observed data set drawn from an unknown distribution, due to two important facts, first, the central limit theorem discussed above, and second, a normal distribution has the maximum entropy or uncertainty among all possible probability density functions with the same variance, which we will prove later.

The univariate normal probability distribution is

$$p(x) = \mathcal{N}(x, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{B.95}$$

with the following mean and variance

$$\mathrm{E}[x] = \mu, \qquad \mathrm{Var}[x] = \sigma^2 \tag{B.96}$$

This is a unimodal bell-shaped curve centered at $x = \mu$, which is the mean, median and mode of the distribution. As the curve is symmetric with respect to μ, its skewness is zero. Its excess kurtosis (Section B.2.4) is zero.

In the multivariate case, the pdf of a normally distributed n-dimensional random vector $\mathbf{x}$ is

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}, \mathbf{m}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}\,|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\mathbf{m})\right) \tag{B.97}$$

with mean vector $\mathbf{m} = \mathrm{E}[\mathbf{x}]$ and covariance matrix $\mathbf{\Sigma} = \mathrm{Cov}[\mathbf{x}]$. We further define the log-normal distribution as:

$$\psi(\mathbf{x}) = \log \mathcal{N}(\mathbf{x}, \mathbf{m}, \mathbf{\Sigma}) = -\frac{1}{2}(\mathbf{x}-\mathbf{m})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{m}) - \frac{n}{2}\log(2\pi) - \frac{1}{2}\log|\mathbf{\Sigma}| \tag{B.98}$$

Its gradient vector and Hessian matrix are:

$$\mathbf{g}_\psi(\mathbf{x}) = \frac{d}{d\mathbf{x}}\psi(\mathbf{x}) = -\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{m}) \tag{B.99}$$

$$\mathbf{H}_\psi(\mathbf{x}) = \frac{d^2}{d\mathbf{x}^2}\psi(\mathbf{x}) = -\mathbf{\Sigma}^{-1} \tag{B.100}$$

Setting the first expression to zero and solving the resulting equation we get the mode $\mathbf{x} = \mathbf{m}$ at which $\mathcal{N}(\mathbf{x}, \mathbf{m}, \mathbf{\Sigma})$ reaches maximum. Solving the second equation for $\mathbf{\Sigma}$ we get the covariance:

$$\mathbf{\Sigma} = -\mathbf{H}_\psi^{-1}(\mathbf{x}) \tag{B.101}$$

The iso-hypersurface of the normal distribution is represented by the following equation:

$$\mathcal{N}(\mathbf{x}, \mathbf{m}, \mathbf{\Sigma}) = c_0 \tag{B.102}$$

where c_0 is a constant. Taking the log on both sides of the equation and ignore the constant scaling factor, we get an equivalent quadratic equation:

$$(\mathbf{x}-\mathbf{m})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{m}) = c_1 \tag{B.103}$$

This equation in general represents a quadratic (e.g., elliptic, parabolic, hyperbolic) hypersurface in a high dimensional space, depending on the definiteness of matrix $\mathbf{\Sigma}$. Here as $\mathbf{\Sigma}$ is positive definite, the equation represents an ellipsoid centered at $\mathbf{m}$, and its orientation in the space is determined by $\mathbf{\Sigma}$. Specially if components in $\mathbf{x}$ are uncorrelated, i.e., $\sigma_{ij} = 0$ for all $i \neq j$, then $\mathbf{\Sigma} = \mathrm{diag}[\sigma_1^2, \cdots, \sigma_n^2]$ is a diagonal matrix and Eq. (B.103) becomes

$$(\mathbf{x}-\mathbf{m})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{m}) = \sum_{i=1}^{n}\frac{(x_i-\mu_i)^2}{\sigma_i^2} = c_1 \tag{B.104}$$

which represents a standard ellipsoid with all its principal semi-axes are in parallel to those of the coordinate system. Still more specially if $\mathbf{x}$ is standardized with $\mathbf{m} = \mathbf{0}$ and $\mathbf{\Sigma} = \mathbf{I}$, then Eq. (B.103) represents a hypersphere.

Properties of Gaussian distribution

- Affine transformation of normal distributions:

  If $p(\mathbf{x}) = \mathcal{N}(\mathbf{m}_x, \mathbf{\Sigma}_x)$, then $p(\mathbf{y}) = p(\mathbf{A}\mathbf{x}+\mathbf{b}) = \mathcal{N}(\mathbf{m}_y, \mathbf{\Sigma}_y)$, where

  $$\mathbf{m}_y = \mathbf{A}\mathbf{m}_x + \mathbf{b}, \qquad \mathbf{\Sigma}_y = \mathbf{A}\mathbf{\Sigma}_x\mathbf{A}^T \tag{B.105}$$

  Proof:

  $$\mathbf{m}_y = \mathrm{E}[\mathbf{A}\mathbf{x}+\mathbf{b}] = \mathbf{A}\ \mathrm{E}[\mathbf{x}] + \mathbf{b} = \mathbf{A}\mathbf{m}_x + \mathbf{b} \tag{B.106}$$

$$\begin{aligned}\boldsymbol{\Sigma}_y &= \mathrm{E}[(\mathbf{y}-\mathbf{m}_y)(\mathbf{y}-\mathbf{m}_y)^T] \\ &= \mathrm{E}[[(\mathbf{A}\mathbf{x}-\mathbf{b})-(\mathbf{A}\mathbf{m}_x-\mathbf{b})][(\mathbf{A}\mathbf{x}-\mathbf{b})-(\mathbf{A}\mathbf{m}_x+\mathbf{b})]^T] \\ &= \mathrm{E}[(\mathbf{A}\mathbf{x}-\mathbf{A}\mathbf{m}_x)(\mathbf{A}\mathbf{x}-\mathbf{A}\mathbf{m}_x)^T] = \mathbf{A}\,\mathrm{E}[(\mathbf{x}-\mathbf{m}_x)(\mathbf{x}-\mathbf{m}_x)^T]\mathbf{A}^T \\ &= \mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T \end{aligned} \tag{B.107}$$

Some special cases:

- If $\mathbf{A}=\mathbf{I}$, then $p(\mathbf{x}+\mathbf{b})=\mathcal{N}(\mathbf{m}_x+\mathbf{b},\boldsymbol{\Sigma}_x)$
- If $\mathbf{b}=\mathbf{0}$, then $p(\mathbf{A}\mathbf{x})=\mathcal{N}(\mathbf{m}_x,\mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T)$
- If $p(\mathbf{x})=N\left([\mu_1,\cdots,\mu_n]^T,\,\mathrm{diag}[\sigma_1^2,\cdots,\sigma_n^2]\right)$, then $p(\sum_{i=1}^n x_i)=\mathcal{N}(\sum_{i=1}^n\mu_i,\sum_{i=1}^n\sigma_i^2)$

- Marginal distributions:

$$p(x_1,x_2)=p(\mathbf{x})=p\left(\begin{bmatrix}x_1\\x_2\end{bmatrix}\right)=N\left(\begin{bmatrix}\mu_1\\\mu_2\end{bmatrix},\begin{bmatrix}\sigma_1^2 & \sigma_{12}^2\\ \sigma_{21}^2 & \sigma_2^2\end{bmatrix}\right) \tag{B.108}$$

Applying $\mathbf{A}_1=[1,\,0]$ and $\mathbf{A}_2=[0,\,1]$ to $\mathbf{x}=[x_1,\,x_2]^T$, we get the two marginal distributions for $\mathbf{A}_1\mathbf{x}=x_1$ and $\mathbf{A}_2\mathbf{x}=x_2$, respectively:

$$p(x_1)=\mathcal{N}(\mu_1,\sigma_1^2),\qquad p(x_1)=\mathcal{N}(\mu_2,\sigma_2^2) \tag{B.109}$$

- Product of Gaussians: Consider two Gaussians:

$$\begin{aligned} p_i(\mathbf{x}) &= \mathcal{N}(\mathbf{x},\mathbf{m}_i,\boldsymbol{\Sigma}_i)=\frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}_i|^{1/2}}\exp\left[-\frac{1}{2}(\mathbf{x}-\mathbf{m}_i)^T\boldsymbol{\Sigma}_i^{-1}(\mathbf{x}-\mathbf{m}_i)\right] \\ &= \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}_i|^{1/2}}\exp\left[-\frac{1}{2}(\mathbf{x}^T\boldsymbol{\Sigma}_i^{-1}\mathbf{x}-2\mathbf{x}^T\boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i+\mathbf{m}_i\boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i)\right] \\ &= \exp\left[-\frac{1}{2}[n\log(2\pi)+\log|\boldsymbol{\Sigma}_i|+\mathbf{m}_i\boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i]\right]\exp\left[-\frac{1}{2}(\mathbf{x}^T\boldsymbol{\Sigma}_i^{-1}\mathbf{x}-2\mathbf{x}^T\boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i)\right] \\ &= \exp\left[-\frac{1}{2}[\mathbf{x}^T\boldsymbol{\Sigma}_i^{-1}\mathbf{x}-2\mathbf{x}^T\boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i+c_i]\right] \qquad (i=1,2) \end{aligned} \tag{B.110}$$

where

$$c_i=n\log(2\pi)+\log|\boldsymbol{\Sigma}_i|+\mathbf{m}_i\boldsymbol{\Sigma}_i^{-1}\mathbf{m}_i \tag{B.111}$$

The product of two Gaussians is

$$\begin{aligned} p_1(\mathbf{x})p_2(\mathbf{x}) &= \mathcal{N}(\mathbf{x},\mathbf{m}_1,\boldsymbol{\Sigma}_2)\mathcal{N}(\mathbf{x},\mathbf{m}_2,\boldsymbol{\Sigma}_2) \\ &= \exp\left[-\frac{1}{2}[\mathbf{x}^T\boldsymbol{\Sigma}_1^{-1}\mathbf{x}-2\mathbf{x}^T\boldsymbol{\Sigma}_1^{-1}\mathbf{m}_1+c_1+\mathbf{x}^T\boldsymbol{\Sigma}_2^{-1}\mathbf{x}-2\mathbf{x}^T\boldsymbol{\Sigma}_2^{-1}\mathbf{m}_2+c_2]\right] \\ &= \exp\left[-\frac{1}{2}[\mathbf{x}^T(\boldsymbol{\Sigma}_1^{-1}+\boldsymbol{\Sigma}_2^{-1})\mathbf{x}-2\mathbf{x}^T(\boldsymbol{\Sigma}_1^{-1}\mathbf{m}_1+\boldsymbol{\Sigma}_2^{-1}\mathbf{m}_2)+c_1+c_2]\right] \\ &= \exp\left[-\frac{1}{2}[\mathbf{x}^T\boldsymbol{\Sigma}^{-1}\mathbf{x}-2\mathbf{x}^T\boldsymbol{\Sigma}^{-1}\mathbf{m}+c+(c_1+c_2-c)]\right] \\ &= \mathcal{N}(\mathbf{x},\mathbf{m},\boldsymbol{\Sigma})\ \exp(c_1+c_2-c)\propto\mathcal{N}(\mathbf{x},\mathbf{m},\boldsymbol{\Sigma}) \end{aligned} \tag{B.112}$$

Here we have defined

$$c=n\log(2\pi)+\log|\boldsymbol{\Sigma}|+\mathbf{m}^T\boldsymbol{\Sigma}^{-1}\mathbf{m} \tag{B.113}$$

$$\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1})^{-1} \tag{B.114}$$

and

$$\boldsymbol{\Sigma}^{-1}\mathbf{m} = \boldsymbol{\Sigma}_1^{-1}\mathbf{m}_1 + \boldsymbol{\Sigma}_2^{-1}\mathbf{m}_2 \tag{B.115}$$

i.e.,

$$\mathbf{m} = \boldsymbol{\Sigma}(\boldsymbol{\Sigma}_1^{-1}\mathbf{m}_1 + \boldsymbol{\Sigma}_2^{-1}\mathbf{m}_2) = (\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1})^{-1}(\boldsymbol{\Sigma}_1^{-1}\mathbf{m}_1 + \boldsymbol{\Sigma}_2^{-1}\mathbf{m}_2) \tag{B.116}$$

We see that the product of Gaussians is also a scaled Gaussian.

- **Marginal and conditional distributions of multivariate normal distribution**

  A random vector $\mathbf{x} = [\mathbf{x}_1^T, \mathbf{x}_2^T]^T$ with a normal distribution $\mathcal{N}(\mathbf{x}, \mathbf{m}, \boldsymbol{\Sigma})$ can be partitioned into two subvectors $\mathbf{x}_1$ and $\mathbf{x}_2$:

$$p\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}\right) = p(\mathbf{x}) = \mathcal{N}(\mathbf{x}, \mathbf{m}, \boldsymbol{\Sigma}) = N\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}, \begin{bmatrix} \mathbf{m}_1 \\ \mathbf{m}_2 \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}\right) \tag{B.117}$$

  where $\boldsymbol{\Sigma}_{ij} = \boldsymbol{\Sigma}_{ji}^T$, $\boldsymbol{\Sigma}_{ii} = \boldsymbol{\Sigma}_{ii}^T$, $(i, j = 1, 2,\ i \neq j)$. Given this normal joint distribution of $\mathbf{x}_1$ and $\mathbf{x}_2$, we can find the normal conditional distribution of $\mathbf{x}_i$ given $\mathbf{x}_j$:

$$p(\mathbf{x}_i/\mathbf{x}_j) = \mathcal{N}(\mathbf{x}_i, \mathbf{m}_{i/j}, \boldsymbol{\Sigma}_{i/j}) \tag{B.118}$$

  in terms of the mean vector and the covariance matrix:

$$\begin{cases} \mathbf{m}_{i/j} &= \mathbf{m}_i + \boldsymbol{\Sigma}_{ij}\boldsymbol{\Sigma}_{jj}^{-1}(\mathbf{x}_j - \mathbf{m}_j) \\ \boldsymbol{\Sigma}_{i/j} &= \boldsymbol{\Sigma}_{ii} - \boldsymbol{\Sigma}_{ij}\boldsymbol{\Sigma}_{jj}^{-1}\boldsymbol{\Sigma}_{ji} \end{cases} \tag{B.119}$$

  Proof: The joint distribution of $\mathbf{x}$ can be written as:

$$\begin{aligned} p(\mathbf{x}) = p(\mathbf{x}_1, \mathbf{x}_2) &= \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \mathbf{m})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \mathbf{m})\right] \\ &= \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}Q(\mathbf{x}_1, \mathbf{x}_2)\right] \end{aligned} \tag{B.120}$$

  Here we have defined:

$$\begin{aligned} Q(\mathbf{x}_1, \mathbf{x}_2) &= (\mathbf{x} - \mathbf{m})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \mathbf{m}) \\ &= [(\mathbf{x}_1 - \mathbf{m}_1)^T, (\mathbf{x} - \mathbf{m}_2)^T] \begin{bmatrix} \boldsymbol{\Sigma}^{11} & \boldsymbol{\Sigma}^{12} \\ \boldsymbol{\Sigma}^{21} & \boldsymbol{\Sigma}^{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 - \mathbf{m}_1 \\ \mathbf{x}_2 - \mathbf{m}_2 \end{bmatrix} \\ &= (\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}^{11}(\mathbf{x}_1 - \mathbf{m}_1) + 2(\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}^{12}(\mathbf{x}_2 - \mathbf{m}_2) + (\mathbf{x}_2 - \mathbf{m}_2)^T\Sigma^{22}(\mathbf{x}_2 - \mathbf{m}_2) \end{aligned} \tag{B.121}$$

  where we have assumed

$$\boldsymbol{\Sigma}^{-1} = \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{\Sigma}^{11} & \boldsymbol{\Sigma}^{12} \\ \boldsymbol{\Sigma}^{21} & \boldsymbol{\Sigma}^{22} \end{bmatrix} \tag{B.122}$$

According to the Woodbury matrix identify (Section A.2.4), we have

$$\begin{aligned}
\boldsymbol{\Sigma}^{11} &= (\boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21})^{-1} = \boldsymbol{\Sigma}_{11}^{-1} + \boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}\boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1} \\
\boldsymbol{\Sigma}^{22} &= (\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1} = \boldsymbol{\Sigma}_{22}^{-1} + \boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}(\boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21})^{-1}\boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1} \\
\boldsymbol{\Sigma}^{12} &= -\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1} = (\boldsymbol{\Sigma}^{12})
\end{aligned} \tag{B.123}$$

Substituting the second expression for $\boldsymbol{\Sigma}^{11}$, the first expression for $\boldsymbol{\Sigma}^{22}$, and $\boldsymbol{\Sigma}^{12}$ into $Q(\mathbf{x}_1, \mathbf{x}_2)$ we get:

$$\begin{aligned}
Q(\mathbf{x}_1, \mathbf{x}_2) &= (\mathbf{x}_1 - \mathbf{m}_1)^T[\boldsymbol{\Sigma}_{11}^{-1} + \boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}\boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}](\mathbf{x}_1 - \mathbf{m}_1) \\
&\quad -2(\mathbf{x}_1 - \mathbf{m}_1)^T[\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}](\mathbf{x}_2 - \mathbf{m}_2) \\
&\quad +(\mathbf{x}_2 - \mathbf{m}_2)^T[(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}](\mathbf{x}_2 - \mathbf{m}_2) \\
&= (\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1) \\
&\quad +(\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}\boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1) \\
&\quad -2(\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}(\mathbf{x}_2 - \mathbf{m}_2) \\
&\quad +(\mathbf{x}_2 - \mathbf{m}_2)^T(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}(\mathbf{x}_2 - \mathbf{m}_2) \\
&= (\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1) \\
&\quad +[(\mathbf{x}_2 - \mathbf{m}_2) - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1)]^T(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}[(\mathbf{x}_2 - \mathbf{m}_2) - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-}
\end{aligned}$$

The last equal sign is due to the following equations for any vectors u and v and a symmetric matrix $A = A^T$:

$$\begin{aligned}
&u^TAu - 2u^TAv + v^TAv = u^TAu - u^TAv - u^TAv + v^TAv \\
&= u^TA(u - v) - (u - v)^TAv = u^TA(u - v) - v^TA(u - v) \\
&= (u - v)^TA(u - v) = (v - u)^TA(v - u)
\end{aligned} \tag{B.125}$$

We further define

$$\begin{cases} \mathbf{b} = \mathbf{m}_2 + \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1) \\ \mathbf{A} = \boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12} \end{cases} \tag{B.126}$$

and

$$\begin{cases} Q_1(\mathbf{x}_1) &= (\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1) \\ Q_2(\mathbf{x}_1, \mathbf{x}_2) &= [(\mathbf{x}_2 - \mathbf{m}_2) - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1)]^T(\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12})^{-1}[(\mathbf{x}_2 - \mathbf{m}_2) - \boldsymbol{\Sigma} \\ &= (\mathbf{x}_2 - \mathbf{b})^T\mathbf{A}^{-1}(\mathbf{x}_2 - \mathbf{b}) \end{cases} \tag{B.127}$$

then the expression above for $Q(\mathbf{x}_1, \mathbf{x}_2)$ can be written as:

$$Q(\mathbf{x}_1, \mathbf{x}_2) = Q_1(\mathbf{x}_1) + Q_2(\mathbf{x}_1, \mathbf{x}_2) \tag{B.128}$$

and the joint distribution can be written as:

$$\begin{aligned} p(\mathbf{x}) = p(\mathbf{x}_1, \mathbf{x}_2) &= \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}Q(\mathbf{x}_1, \mathbf{x}_2)\right] \\ &= \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}_{11}|^{1/2}|\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}|^{1/2}} \exp\left[-\frac{1}{2}Q_1(\mathbf{x}_1)\right] \exp\left[-\frac{1}{2}Q_2(\mathbf{x}_1, \mathbf{x}_2)\right] \\ &= \frac{1}{(2\pi)^{p/2}|\boldsymbol{\Sigma}_{11}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1)\right] \frac{1}{(2\pi)^{q/2}|\mathbf{A}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}_2 - \mathbf{b})^T\mathbf{A}^{-1}(\mathbf{x}_2 - \mathbf{b}\right. \\ &= \mathcal{N}(\mathbf{x}_1, \mathbf{m}_1, \boldsymbol{\Sigma}_{11})\, \mathcal{N}(\mathbf{x}_2, \mathbf{b}, \mathbf{A}) \end{aligned} \tag{B.1}$$

where we have used theorem 2 in Section 7 in Appendix A:

$$|\boldsymbol{\Sigma}| = |\boldsymbol{\Sigma}_{11}|\, |\boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}| \tag{B.130}$$

The marginal distribution of $\mathbf{x}_1$ is

$$\begin{aligned} p_1(\mathbf{x}_1) &= \int p(\mathbf{x}_1, \mathbf{x}_2) d\mathbf{x_2} = \mathcal{N}(\mathbf{x}_1, \mathbf{m}_1, \boldsymbol{\Sigma}_{11}) \int \mathcal{N}(\mathbf{x}_2, \mathbf{b}, \mathbf{A}) d\mathbf{x_2} \\ &= \mathcal{N}(\mathbf{x}_1, \mathbf{m}_1, \boldsymbol{\Sigma}_{11}) = \frac{1}{(2\pi)^{p/2}|\boldsymbol{\Sigma}_{11}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}_1 - \mathbf{m}_1)^T\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1)\right] \end{aligned} \tag{B.131}$$

and the conditional distribution of $\mathbf{x}_2$ given $\mathbf{x}_1$ is

$$p_{2|1}(\mathbf{x}_2|\mathbf{x}_1) = \frac{p(\mathbf{x}_1, \mathbf{x}_2)}{p_1(\mathbf{x}_1)} = \mathcal{N}(\mathbf{x}_2, \mathbf{b}, \mathbf{A}) = \frac{1}{(2\pi)^{q/2}|\mathbf{A}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}_2 - \mathbf{b})^T\mathbf{A}^{-1}(\mathbf{x}_2 - \mathbf{b})\right] \tag{B.132}$$

with

$$\begin{cases} \mathbf{b} &= \mathbf{m}_{2/1} = \mathbf{m}_2 + \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\mathbf{x}_1 - \mathbf{m}_1) \\ \mathbf{A} &= \boldsymbol{\Sigma}_{2/1} = \boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12} \end{cases} \tag{B.133}$$

Drawing samples from Gaussian distributions

We first consider drawing samples from a random variable x with a normal distribution $p(x) = \mathcal{N}(0, 1)$, i.e., $\mathrm{E}[x] = 0$ and $\mathrm{Var}[x] = \mathrm{E}[x^2] = 1$. As shown in the figure, a random variable uniformly distributed in the range $(0,\ 1)$ (vertical) can be mapped to another random variable x in the range $(-\infty, \infty)$ (horizontal) with a Gaussian distribution $p(x) = \mathcal{N}(0,\ 1)$, by the inverse function of its accumulative $F(x) = \int_{-\infty}^{x} p(u)\, du$. This variable x can be further converted into another variable $x' = \mu + \sigma x$, whose distribution is a generic Gaussian $p(x') = \mathcal{N}(\mu,\ \sigma^2)$, with the mean and variance

$$\begin{aligned} \mathrm{E}[x'] &= \mathrm{E}[\mu + \sigma x] = \mu + \sigma\, \mathrm{E}[x] = \mu \\ \mathrm{Var}[x'] &= [(x' - \mu)^2] = \mathrm{E}[\sigma^2 x] = \mathrm{E}[\sigma^2 x^2] = \sigma^2\, \mathrm{E}[x^2] = \sigma^2 \end{aligned} \tag{B.134}$$

Using this method, we can generate a set of n normally distributed and independent random variables $x_1, \cdots, x_n$, each with $\mathrm{E}[x_i] = 0$, $Var(x_i) = 1$, and $\mathrm{Cov}[x_i, x_j] = \delta_{ij}$, and, based on these variables, we can further construct a random vector $\mathbf{x} = [x_1, \cdots, x_n]^T$ with a Gaussian joint distributed $p(\mathbf{x}) = \mathcal{N}(\mathbf{0},\ \mathbf{I})$.

Given such an $\mathbf{x}$ with $\mathbf{m}_x = \mathbf{0}$ and $\mathbf{\Sigma}_x = \mathbf{I}$, we can generate another random vector with a Gaussian distribution $N(\mathbf{m}, \mathbf{\Sigma})$. To do so, consider the affine transformation $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{m}$, which also has a Gaussian distribution:

$$\begin{aligned} \mathbf{m}_y &= \mathrm{E}[\mathbf{y}] = \mathbf{W}\mathbf{m}_x + \mathbf{m} = \mathbf{m} \\ \mathbf{\Sigma}_y &= \mathbf{W}\mathbf{\Sigma}_x\mathbf{W}^T = \mathbf{W}\mathbf{W}^T \end{aligned} \tag{B.135}$$

While $\mathbf{m}_y = \mathbf{m}$ as desired, we still need to make sure that $\mathbf{\Sigma}_y = \mathbf{W}\mathbf{W}^T = \mathbf{\Sigma}$ also as desired. There exist infinite matrices all satisfying the equation above. Typically, such a $\mathbf{W}$ can be found in either of the following two methods:

- The Cholesky decomposition: which can find an upper triangular matrix $\mathbf{L}$ that satisfies $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^T$ (if $\mathbf{\Sigma}$ is nonsingular).
- The eigenvalue decomposition $\mathbf{\Sigma} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$ (even if $\mathbf{\Sigma}$ is singular), where $\mathbf{V}$ and $\mathbf{\Lambda}$ are respectively the eigenvector and eigenvalue matrices of $\mathbf{\Sigma}$. As $\mathbf{\Sigma}^T = \mathbf{\Sigma}$ is symmetric and positive definite, its eigenvalues take positive real values and its eigenvectors are orthogonal to each other, i.e., $\mathbf{V}^T = \mathbf{V}^{-1}$ is orthogonal. If we let $\mathbf{W} = \mathbf{V}\mathbf{\Lambda}^{1/2}$, then we have

$$\mathbf{W}\mathbf{W}^T = (\mathbf{V}\mathbf{\Lambda}^{1/2})\,(\mathbf{V}\mathbf{\Lambda}^{1/2})^T = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T = \mathbf{\Sigma} \tag{B.136}$$

We note that although $\mathbf{L}\mathbf{L}^T = \mathbf{W}\mathbf{W}^T = \mathbf{\Sigma}$, $\mathbf{L}$ found by the Cholesky decomposition method is upper triangular but $\mathbf{W}$ is not, $\mathbf{L} \neq \mathbf{W}$. Also, we note that the transform based on eigenvalue decomposition is optimal in certain way, as to be discussed later.

Having found either L or $\mathbf{W}$, we get $\mathbf{y} = \mathbf{L}\mathbf{x} + \mathbf{m}$ or $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{m}$ with $p(\mathbf{y}) = \mathcal{N}(\mathbf{m}, \mathbf{\Sigma})$ as desired.

B.1.6 Chi-Square and t-Distributions

Let $\{x_1, \cdots, x_N\}$ be a set of N i.i.d. samples from a normal distribution $x_n \sim \mathcal{N}(\mu, \sigma^2)$. The mean μ and variance σ^2 can be estimated by

$$\hat{\mu}_N = \frac{1}{N}\sum_{n=1}^{N} x_n, \qquad \hat{\sigma}_N^2 = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \hat{\mu})^2 \tag{B.137}$$

The estimated mean $\hat{\mu}$ and variance $\hat{\sigma}^2$ are in turn random variables associated with two important probability density distributions:

- Student's t-distribution:

  The standardized estimated mean in Eq. (B.137) has a standard normal distribution:

$$\frac{\hat{\mu}_N - \mu}{\sigma/\sqrt{N}} \;\sim\; \mathcal{N}(0,\ 1) \tag{B.138}$$

When the true standard deviation σ is unknown and replaced by its estimation $\hat{\sigma}$, then the standardized estimated mean has a Student's t-distribution:

$$t = \frac{\hat{\mu}_N - \mu}{\hat{\sigma}/\sqrt{N}} \sim \mathcal{T}_\nu(t) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\,\Gamma\left(\frac{\nu}{2}\right)}\left(1+\frac{t^2}{\nu}\right)^{-(\nu+1)/2} \tag{B.139}$$

where $\nu = N-1$ is the degree of freedom and $\Gamma(n) = (n-1)!$ is the gamma function, with zero mean and variance $\nu/(\nu-2)$. Specially when $\nu = 1$, $\Gamma((\nu+1)/2) = \Gamma(1) = 1$, $\Gamma(\nu/2) = \Gamma(1/2) = \sqrt{\pi}$, and

$$\mathcal{T}_1 = \frac{1}{\pi}(1+t^2)^{-1} \tag{B.140}$$

When $\nu \to \infty$, the t-distribution approaches the standard normal distribution:

$$\mathcal{T}_\nu \overset{\nu\to\infty}{\Longrightarrow} \mathcal{N}(0,1) \tag{B.141}$$

We can further get the distribution of variable $\hat{\mu}_N - \mu$ (Eq. (B.71)):

$$\hat{\mu}_N - \mu = \frac{\sigma}{\sqrt{N}}\, t \sim \frac{\sqrt{N}}{\sigma}\mathcal{T}_\nu(t) \tag{B.142}$$

with zero mean and variance σ^2/N (Eq. (B.75)), i.e., $\hat{\mu}_N$ has mean μ and variance σ^2/N.

When $\nu = N-1 \to \infty$, the variance of this distribution approaches zero, i.e., the estimated $\hat{\mu}$ approaches the true μ:

$$\hat{\mu}_N = \frac{1}{N}\sum_{n=1}^{N} x_n = \frac{\sigma}{\sqrt{N}}\, t - \mu \overset{N\to\infty}{\Longrightarrow} \mu \tag{B.143}$$

- Chi-square distribution:

  A random variable $x \sim \mathcal{N}(\mu, \sigma^2)$ can be standardized to become

$$z = \frac{x-\mu}{\sigma} \sim \mathcal{N}(0,1) \tag{B.144}$$

  and the sum of squares of a set of N i.i.d. samples $\{z_1, \cdots, z_N\}$ of z has a chi-square distribution:

$$s = \sum_{n=1}^{N} z_n^2 = \sum_{n=1}^{N}\left(\frac{x_n-\mu}{\sigma}\right)^2 \sim \chi^2(s) = \frac{s^{\nu/2-1}e^{-s/2}}{2^{\nu/2}\Gamma(\nu/2)} \tag{B.145}$$

  where $\nu = N-1$ is the degree of freedom, with mean ν and variance 2ν. When $\nu \to \infty$, the chi-square distribution tends to a normal distribution $\mathcal{N}(\nu, 2\nu)$, which has infinite variance and approaches 0.

  The estimated variance $\hat{\sigma}^2$ in Eq. (B.137) can be written as (Eq. (B.71)):

$$\hat{\sigma}^2 = \frac{1}{N-1}\sum_{n=1}^{N}(x_n-\hat{\mu})^2 = \frac{\sigma^2}{N-1}\, s = \frac{\sigma^2}{\nu}\, s \sim \frac{\nu}{\sigma^2}\,\chi^2 \tag{B.146}$$

with the following mean and variance (Eq. (B.75)):

$$\frac{\sigma^2}{\nu}\,\nu = \sigma^2, \qquad \left(\frac{\sigma^2}{\nu}\right)^2 2\nu = 2\sigma^4/\nu \tag{B.147}$$

When $\nu = N - 1 \to \infty$, the variance of the distribution approaches zero, i.e., the estimated $\hat{\sigma}^2$ approaches the true σ^2:

$$\hat{\sigma}_N^2 = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \hat{\mu})^2 = \frac{\sigma^2}{\nu}\; s \overset{N\to\infty}{\Longrightarrow} \sigma^2 \tag{B.148}$$

B.2 Entropy and Information

B.2.1 Entropy

Entropy is a measurement of the uncertainty of the outcome of a random event, which also measures the amount of information obtained in terms of the reduction of uncertainty (due to certain communication or experiment). If the uncertainty is reduced to zero by a communication, then the amount of information obtained is exactly the same as the uncertainty before the communication.

A random event of n outcomes can be represented by a discrete random variable x with N possible values $\{x_1, \cdots, x_n\}$ with the corresponding probabilities $P_i = P(x_i)$ satisfying $\sum_{i=1}^{n} P_i = 1$. For example, $n = 26$ letters in the English alphabet, with different probabilities to be used (e.g., $P(x =' a') > P(x =' z')$).

To quantitatively measure the *uncertainty* of random variable x denoted by $H(x)$, we define the uncertainty or information content (surprise) of each outcome x_i as

$$H(x_i) = \log\frac{1}{P_i} = -\log P_i, \qquad (i = 1, \cdots, n) \tag{B.149}$$

so that the desired properties listed below are satisfied:

- Uncertainty is inversely related to the probability, i.e., a more probable outcome is less uncertain than a less probable outcome. If $P_i > P_j$, then

$$H(x_i) = -\log P_i < H(x_j) = \log P_j \tag{B.150}$$

- Specially, an certain outcome x with $P(x_i) = 1$ has zero uncertainty, and an impossible outcome x with $P(x_j) = 0$ has infinite uncertainty:

$$H(x_i) = -\log P_i = -\log 1 = 0, \qquad H(x_j) = -\log P_j = -\log 0 = \infty \tag{B.151}$$

- The total uncertainty of x is the expectation of the uncertainties of the n outcomes:

$$H(x) = \mathrm{E}[H(x_i)] = \mathrm{E}\left[-\log P_i\right] = -\sum_{i=1}^{n} P_i \log P_i \tag{B.152}$$

The total uncertainty $H(x)$ defined above, called the *entropy*, measures the total uncertainty of the random event represented by x, which is also the total amount of information contained in x. Once the outcome of the random event is known, the information contained in x is gained, and the uncertainty is reduced to zero.

The n outcomes of x can be binary encoded so that each outcome x_i is represented by a sequence of 0's and 1's. To minimizes the total number of bits needed to encode x, we encode x_i with P_i by $H(x_i) = -\log_2 P_i$ bits, so that a more probable outcome x_i with greater P_i (less uncertain and containing less information) is encoded by fewer bits, and a less probable outcome x_j with smaller P_j (more uncertain and containing more information) is encoded by more bits. Then the expected number of bits needed to encode x is minimized.

We can show that a uniform distribution of $n = 2^m$ equally likely outcomes with $p_i = 1/n$ $(i = 1, \cdots, n)$ has the maximum entropy. This is a constrained maximization problem which can be solved by Lagrange multiplier method:

$$\frac{\partial}{\partial P_j}\left[-\sum_{i=1}^{n} P_i log\ P_i + \lambda\left(1 - \sum_{i=0}^{n} P_i\right)\right] = 0$$

$$= -\frac{\partial}{\partial P_j} P_j\ log\ P_j - \lambda = -(log\ P_j + 1) - \lambda = 0 \tag{B.153}$$

Solving this we get

$$P_j = 2^{-\lambda - 1} \qquad (j = 1, \cdots, N) \tag{B.154}$$

which must satisfy

$$\sum_{j=1}^{n} P_j = \sum_{j=1}^{n} 2^{-\lambda-1} = 2^{-\lambda-1} n = 1 \tag{B.155}$$

Solving for λ we get

$$\lambda = \log n - 1 \tag{B.156}$$

Substituting back into P_j we get

$$P_j = 2^{-\lambda-1} = 2^{-\log n} = \frac{1}{n}, \qquad (j = 1, \cdots, n) \tag{B.157}$$

The corresponding maximum entropy can be obtained as

$$H_{max}(E_1, \cdots, E_n) = -\sum_{i=1}^{n} \frac{1}{n} log_2 \frac{1}{n} = log_2 n = m \tag{B.158}$$

In this case, each outcome x_i is encoded by

$$\log_2 \frac{1}{P_i} = \log_2 \left(\frac{1}{1/2^n}\right) = \log_2 2^n = n \tag{B.159}$$

bits, and the total number of bits needed to encode this random event is

$$-\sum_{i=1}^{n} P_i \log_2 P = -\sum_{i=1}^{n} \frac{1}{n} \log_2 \frac{1}{2^m} = \sum_{i=1}^{n} \frac{1}{n} \log_2 2^m = m \tag{B.160}$$

If $n = 2^m = 2^3 = 8$, $P_i = 1/2^3 = 1/8$, $n = 3$ bits are needed to encode the $n = 8$ possible outcomes.

For example, in the special case of $n = 2$ outcomes:

- If $P_1 = 1$ and $P_2 = 0$, we have the minimum uncertainty zero ($x \log_2 x|_{x=0} = 0$), i.e., 0 bit is needed to encode a sure event:

$$H = -1\ log_2\ 1 - 0\ log_2\ 0 = 0 \tag{B.161}$$

- If $P_1 = 0.1$, $P_2 = 0.9$, the uncertainty is:

$$H = -(0.1\ log_2\ 0.1 + 0.9\ log_2\ 0.9) = 0.47 \tag{B.162}$$

- If $P_1 = 0.2$, $P_2 = 0.8$, the uncertainty is:

$$H = -(0.2\ log_2\ 0.2 + 0.8\ log_2\ 0.8) = 0.72 \tag{B.163}$$

- If $P_1 = 0.3$, $P_2 = 0.7$, the uncertainty is:

$$H = -(0.3\ log_2\ 0.3 + 0.7\ log_2\ 0.7) = 0.88 \tag{B.164}$$

- If $P_1 = 0.4$, $P_2 = 0.6$, the uncertainty is:

$$H = -(0.4\ log_2\ 0.4 + 0.6\ log_2\ 0.6) = 0.97 \tag{B.165}$$

- If $P_1 = P_2 = 0.5 = 1/2$, we have the maximum uncertainty $H = 1$, i.e., 1 bit (0 or 1) is needed to encode the event:

$$H = -\frac{1}{2}\ log_2\ \frac{1}{2} - \frac{1}{2}\ log_2\ \frac{1}{2} = \frac{1}{2} + \frac{1}{2} = 1 \tag{B.166}$$

The discussion above for a discrete random variable x can be extended to a continuous random variable x with pdf $p(x)$, of which the *differential entropy* is defined as

$$H(p) = -\int p(x) \log p(x)\ dx \tag{B.167}$$

The *perplexity* of a discrete probability distribution P is simply defined as $2^{H(p)}$.

We consider the pdf $p(x)$ with maximum entropy under each of the following two conditions:

- The distribution $p(x)$ is non-zero only inside a closed region $x \in [a,\ b]$.

  To find the pdf $p(x)$ that maximizes $H(p)$ under the condition $\int_a^b p(x)\,dx = 1$, we construct its Lagrangian function

$$L(p) = H(p) + \lambda \left(1 - \int_a^b p(x)\,dx\right) = \int_a^b \left[-p(x)\ \log p(x) - \lambda p(x)\right]\ dx + \lambda \tag{B.168}$$

  and set its derivative with respect to $p(x)$ (with a specific value of x) to zero:

$$\frac{d\,L(p)}{d\,p(x)} = -(\log\ p(x) + 1) - \lambda = 0 \tag{B.169}$$

Solving for $p(x)$ we get

$$p(x) = e^{-\lambda - 1} \tag{B.170}$$

which must satisfy

$$\int_{-\infty}^{\infty} p(x)\,dx = \int_{a}^{b} e^{-\lambda-1}\,dx = e^{-\lambda-1}(b-a) = 1 \tag{B.171}$$

i.e.,

$$p(x) = e^{-\lambda-1} = \frac{1}{b-a} \tag{B.172}$$

This is a uniform distribution over $[a,\ b]$. The corresponding maximum entropy is

$$H(p) = -\int_{a}^{b} \frac{1}{b-a} \log\left(\frac{1}{b-a}\right) = \log(b-a) \tag{B.173}$$

- The distribution $p(x)$ is non-zero for all $x \in (-\infty,\ \infty)$ with a known variance

$$\sigma^2 = \int_{-\infty}^{\infty} (x-\mu)^2 p(x)\,dx \tag{B.174}$$

To find the pdf $p(x)$ with σ^2 that maximizes $H(p)$, we consider the following Lagrangian function

$$L(p) = -\int_{-\infty}^{\infty} p(x)\ \log p(x)\,dx + \lambda_1\left(1 - \int_{-\infty}^{\infty} p(x)\,dx\right) + \lambda_2\left(\sigma^2 - \int_{-\infty}^{\infty} (x-\mu)^2\,p(x)\,dx\right) \tag{B.175}$$

and set its derivative with respect to $p(x)$ (evaluated at certain x) to zero:

$$\frac{d\,L(p)}{d\,p(x)} = -(log\,p(x) + 1) - \lambda_1 - \lambda_2(x-\mu)^2 = 0 \tag{B.176}$$

Solving for $p(x)$ we get

$$p(x) = e^{-(\lambda_1 + \lambda_2(x-\mu)^2 + 1)} = e^{-(\lambda_1+1)} e^{-\lambda_2(x-\mu)^2} \tag{B.177}$$

which must satisfy the two constraints:

$$1 = \int_{-\infty}^{\infty} p(x)\,dx = e^{-\lambda_1 - 1} \int_{-\infty}^{\infty} e^{-\lambda_2(x-\mu)^2} dx = e^{-(\lambda_1+1)} \sqrt{\frac{\pi}{\lambda_2}} \tag{B.178}$$

and

$$\begin{aligned} \sigma^2 &= \int_{-\infty}^{\infty} (x-\mu)^2 p(x)\,dx = e^{-\lambda_1-1} \int_{-\infty}^{\infty} e^{-\lambda_2(x-\mu)^2} (x-\mu)^2\,dx \\ &= e^{-\lambda_1-1} \frac{1}{2} \sqrt{\frac{\pi}{\lambda_2^3}} = e^{-\lambda_1-1} \frac{1}{2\lambda_2} \sqrt{\frac{\pi}{\lambda_2}} \end{aligned} \tag{B.179}$$

Comparing the two equations above, we get

$$\lambda_2 = \frac{1}{2\sigma^2} \tag{B.180}$$

Substituting this into Eq. (B.178), we get

$$e^{-(\lambda_1+1)}\sqrt{2\pi\sigma^2} = 1, \qquad \text{i.e.} \qquad e^{-(\lambda_1+1)} = \frac{1}{\sqrt{2\pi\sigma^2}} \tag{B.181}$$

Substituting this together with $\lambda_2 = 1/2\sigma^2$ into $p(x)$ we get

$$p(x) = e^{-(\lambda_1+1)}e^{-\lambda_2(x-\mu)^2} = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} = \mathcal{N}(x, \mu, \sigma^2) \tag{B.182}$$

This is the normal or Gaussian distribution and the corresponding maximum entropy is

$$\begin{aligned} H(g) &= -\int_{-\infty}^{\infty} g(x)\log g(x)\,dx = -\int_{-\infty}^{\infty} g(x)\log\left(\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(x-\mu)^2/2\sigma^2}\right)\,dx \\ &= -\int_{-\infty}^{\infty} g(x)\left(log\frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(x-\mu)^2}{2\sigma^2}\right)dx \\ &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\int_{-\infty}^{\infty}(x-\mu)^2 g(x)\,dx \\ &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sigma^2 = \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2}\log(e) \\ &= \frac{1}{2}\log(2\pi e\sigma^2) = \frac{1}{2}\log(2\pi e) + \log\sigma = 1.4189 + \log\sigma \end{aligned} \tag{B.183}$$

In particular, the entropy of a Gaussian distribution with unit variance $\sigma = 1$ is $H(g) = 1.4189$.

The uncertainty of a distribution $p(x)$ can be measured by both its variance $\sigma^2 = \int(x-\mu)^2 p(x)\ dx$ and its entropy $H = -\int p(x)\log p(x)\ dx$. However, they are also different in that σ^2 represents the spread of $p(x)$ while H captures the shape of $p(x)$. In multimodal distribution (with two or more modes), H is a better measurement as it increases when there exist multiple modes while σ^2 is not sensitive to them.

In data analysis and modeling, if no additional knowledge is available regarding the observed data other than the variance (or covariance for multivariate data), the normal distribution with maximum entropy should be used as least-informative default to model the data, as it will impose the least amount of unsupported constraint and thereby causing minimum bias. This is the *principle of maximum entropy.*

- **Joint Entropy**

  The joint entropy of two discrete random variables x and y is defined as:

$$H(x, y) = -\,\mathrm{E}[\log P(x_i, y_j)] = -\sum_i \sum_j P(x_i, y_j)\log P(x_i, y_j) \tag{B.184}$$

  More generally, the joint entropy of n discrete random variables in $\mathbf{x} =$

$[x_1, \cdots, x_n]^T$ is defined as:

$$\begin{aligned} H(\mathbf{x}) = H(x_1, \cdots, x_n) &= -\,\mathrm{E}[\log P(x_1, \cdots, x_n)] \\ &= -\sum_{x_1} \cdots \sum_{x_n} P(x_1, \cdots, x_n) \log P(x_1, \cdots, x_n) \\ &= -\sum_{x_1} \cdots \sum_{x_n} P(\mathbf{x}) \log P(\mathbf{x}) \end{aligned} \tag{B.185}$$

The differential entropy is

$$\begin{aligned} H(\mathbf{x}) = H(x_1, \cdots, x_n) &= -\int \cdots \int p(x_1, \cdots, x_n) \log P(x_1, \cdots, x_n) dx_1 \cdots dx_n \\ &= -\int p(\mathbf{x}) \log P(\mathbf{x}) d\mathbf{x} \end{aligned} \tag{B.186}$$

- **Entropy of function of random variables**

  Given the entropy $H(\mathbf{x})$ of $\mathbf{x}$, we can further find the entropy of a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ of $\mathbf{x}$. We first recall that the pdf of $\mathbf{y}$ can be expressed in terms of that of $\mathbf{x}$ in Eq. (B.83): $p_Y(\mathbf{y}) = |\det \mathbf{J}_{f^{-1}}| p_X(\mathbf{x})$. We can then find $H(\mathbf{y})$ directly as:

$$\begin{aligned} H(\mathbf{y}) &= -\int p_y(\mathbf{y}) \log p_y(\mathbf{y}) d\mathbf{y} \\ &= -\int |\det \mathbf{J}_{f^{-1}}|\, p_X(\mathbf{x})\ \log \left(|\det \mathbf{J}_{f^{-1}}|\, p_X(\mathbf{x}) \right)\ |\det \mathbf{J}_f|\, d\mathbf{x} \\ &= -\int p_X(\mathbf{x}) \left[\log |\det \mathbf{J}_{f^{-1}}| + \log p_X(\mathbf{x}) \right]\, d\mathbf{x} \\ &= -\int p_X(\mathbf{x}) \log p_X(\mathbf{x})\, d\mathbf{x} - \int p_X(\mathbf{x}) \log |\det \mathbf{J}_{f^{-1}}|\, d\mathbf{x} \\ &= H(\mathbf{x}) + \mathrm{E}\left[\log |\det \mathbf{J}_f|\right] \end{aligned} \tag{B.187}$$

  Here we have used integration by substitution and also the inverse function theorem stating that given the Jacobian $\mathbf{J}_f$ of a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, the Jacobian of the inverse function $\mathbf{x} = \mathbf{f}^{-1}(\mathbf{y})$ is $\mathbf{J}_{f^{-1}} = \mathbf{J}_f^{-1}$. Also as in general $\det(\mathbf{A}^{-1}) = (\det(\mathbf{A}))^{-1}$, we have $|\det \mathbf{J}_{f^{-1}}|\, |\det \mathbf{J}_f| = 1$.

  Specially for a linear function $\mathbf{y} = \mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x}$, we have $\mathbf{J}_f = d\mathbf{f}(\mathbf{x})/d\mathbf{x} = \mathbf{A}$ and $\mathbf{J}_{f^{-1}} = \mathbf{A}^{-1}$, then we have

$$H(\mathbf{y}) = H(\mathbf{x}) + \log |\det \mathbf{A}| \tag{B.188}$$

- **Conditional Entropy**

  The *conditional entropy* $H(y|x)$ measures the uncertainty of a random variable y given the value of another random variable x, or the amount of information gained once the outcome of y is known, given the outcome of x.

Let $H(y|x = x_i)$ be the entropy of y conditioned on $x = x_i$:

$$H(y|x = x_i) = -\sum_j P(y_j|x_i) \log P(y_j|x_i) \tag{B.189}$$

Then the conditional entropy $H(y|x)$ is defined as the average of $H(y|x = x_i)$ over all outcomes of x:

$$\begin{aligned} H(y|x) &= E_x[H(y|x = x_i)] = \sum_i P(x_i) H(y|x = x_i) \\ &= -\sum_i P(x_i) \sum_j P(y_j|x_i) \log P(y_j|x_i) = -\sum_i \sum_j P(x_i) P(y_j|x_i) \log P(y_j|x_i) \\ &= -\sum_i \sum_j P(x_i, y_j) \log P(y_j|x_i) \end{aligned} \tag{B.190}$$

The last equality is due to $P(x, y) = P(y|x)P(x)$.

Theorem:

$$\begin{aligned} H(y|x) &= -\sum_i \sum_j P(x_i, y_j) \log P(y_j|x_i) = \sum_i \sum_j P(x_i, y_j) \log \frac{P(x_i)}{P(x_i, y_j)} \\ &= -\sum_i \sum_j P(x_i, y_j) \log P(x_i, y_j) + \sum_i \sum_j P(x_i, y_j) \log P(x_i) \\ &= H(x, y) + \sum_i P(x_i) \log P(x_i) = H(x, y) - H(x) \end{aligned} \tag{B.191}$$

Similarly, we also have $H(x|y) = H(x, y) - H(y)$. We therefore have

$$H(x, y) = H(y|x) + H(x) = H(x|y) + H(y) \tag{B.192}$$

Substituting this into the previous equation we get *Bayes's rule* for conditional entropy:

$$H(y|x) = H(x|y) + H(y) - H(x) \tag{B.193}$$

B.2.2 Kullback-Leibler Divergence and Mutual Information

- **Cross Entropy**

  When the true distribution P is unknown, the encoding of x can be based on another distribution Q as a model that approximates P. Then the average of the total number of bits needed is called the *cross-entropy*:

$$H(P, Q) = \sum_i P_i \log_2 \frac{1}{Q_i} = -\sum_i P_i \log Q_i = -E_P [\log Q] \tag{B.194}$$

  which can be considered as the expectation of the logarithmic model probability $\log Q_i$ with respect to the unknown ground truth probability P_i.

Gibbs' inequality: The cross entropy $H(P,Q)$ is no smaller than the entropy $H(P)$:

$$H(P) = -\sum_i P_i \log P_i \leq H(P,Q) = -\sum_i P_i \log Q_i \tag{B.195}$$

The equality holds $H(P,Q) = H(P)$ if and only if $Q = P$. For the model Q to be as close to the ground truth P as possible, we need to minimize the cross entropy $H(P,Q)$.

Proof: Based on inequality $\log x \leq x - 1$, we have

$$\sum_i P_i \log\left(\frac{Q_i}{P_i}\right) \leq \sum_i P_i \left(\frac{Q_i}{P_i} - 1\right) = \sum_i Q_i - \sum_i P_i = 0 \tag{B.196}$$

i.e.,

$$\sum_i P_i \log\left(\frac{Q_i}{P_i}\right) = \sum_i P_i \log\left(\frac{1}{P_i}\right) + \sum_i P_i \log Q_i = H(P) - H(P,Q) \leq 0 \tag{B.197}$$

We see that the encoding of x based on model distribution Q always requires more bits than that based on the true distribution P.

- **Kullback-Leibler Divergence (Relative Entropy)**

  The Kullback-Leibler (KL) divergence or relative entropy is the difference between the cross entropy $H(P,Q)$ and the entropy $H(P)$:

$$\begin{aligned} D_{KL}(P||Q) &= H(P,Q) - H(P) = -E_P[\log Q] - E_P[\log P] \\ &= -\sum_i P_i \log Q_i - (-\sum_i P_i \log P_i) \\ &= \sum_i P_i(\log P_i - \log Q_i) = \sum_i P_i \log\left(\frac{P_i}{Q_i}\right) \geq 0 \end{aligned} \tag{B.198}$$

  The KL-divergence represents the number of extra bits needed to encode x based on Q instead of P, or a measure of the error of using Q to approximate P, in terms of the amount of information lost, due to the inaccuracy of the model. In order to obtain the best model Q that optimally approximates P, we need to minimize $D_{KL}(P||Q)$.

  In general, $D_{KL}(P||Q) \neq D_{KL}(Q||P)$, the KL-divergence is not a distance metric as it is not symmetric (hence the name divergence rather than distance). The KL-divergence can be symmetrized to become the Jensen–Shannon (JS) divergence:

$$JS(P||Q) = \frac{1}{2}\left[D_{KL}(P||M) + D_{KL}(Q||M)\right], \qquad \text{where } M = \frac{P+Q}{2} \tag{B.199}$$

  As JS divergence is symmetric, it can be used as a distance measure for the similarity between two distributions P and Q.

  All discussions above for discrete variables can be generalized to continuous variables with summations replaced by integrals.

In particular, consider the cross-entropy of a Gaussian $g(x)$ and an arbitrary pdf $p(x)$ both with the same variance σ^2:

$$\begin{aligned} H(p,g) &= -\int p(x)\log g(x)\,dx = -\int p(x)\left(log\frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(x-\mu)^2}{2\sigma^2}\right)dx \\ &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\int (x-\mu)^2 p(x)\,dx = \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sigma^2 \\ &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2} = \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2}\log e = \frac{1}{2}\log(2\pi e\sigma^2) \quad (\text{B.200}) \end{aligned}$$

We see that the cross-entropy $H(p,g)$ of $g(x)$ and $p(x)$ is the same as the entropy of $g(x)$. But as we also know $H(p,g) \geq H(p)$, we have $H(g) = H(p,g) \geq H(p)$. We therefore conclude that among all pdfs $p(x)$ with the same variance σ^2, the Gaussian $N(x,\mu,\sigma^2)$ has the maximum entropy.

- **Mutual Information**

Mutual information $I(\mathbf{x})$ of all components of a random vector $\mathbf{x} = [x_1,\cdots,x_n]^T$ measures the information shared by its components. It is defined as the KL-divergence between $P(\mathbf{x}) = P(x_1,\cdots,x_n)$ and $\prod_{i=1}^n P(x_j)$:

$$\begin{aligned} I(\mathbf{x}) = I(x_1,\cdots,x_n) &= D_{KL}\left(P(\mathbf{x})||\prod_{j=1}^n P(x_j)\right) \\ &= \sum_{i_1}\cdots\sum_{i_n} P(\mathbf{x})\log\left(\frac{P(\mathbf{x})}{\prod_{j=1}^n P(x_j)}\right) \\ &= \sum_{i_1}\cdots\sum_{i_n} P(\mathbf{x})\log P(\mathbf{x}) - \sum_{i_1}\cdots\sum_{i_n} P(\mathbf{x})\sum_{j=1}^n \log P(x_j) \\ &= -H(\mathbf{x}) - \sum_{j=1}^n\left[\sum_{i_1}\cdots\sum_{i_n} P(\mathbf{x})\log P(x_j)\right] \\ &= \sum_{j=1}^n H(x_j) - H(\mathbf{x}) \end{aligned} \quad (\text{B.201})$$

i.e., the mutual information is the error of using $P(x)\,P(y)$ to model the joint probability $P(x,y)$. When all n variables $x_1,\cdots,x_n$ are independent, i.e., $P(x_1,\cdots,x_n) = \prod_{j=1}^n P(x_j)$, then their mutual information is zero.

Specially the mutual information between two variables X and Y is

$$\begin{aligned} I(x,y) &= \sum_i \sum_j P(x_i,y_j) \log \frac{P(x_i,y_j)}{P(x_i)P(y_j)} \\ &= \sum_i \sum_j P(x_i,y_j) \log \frac{P(x_i,y_j)}{P(x_i)} - \sum_i \sum_j P(x_i,y_j) \log P(y_j) \\ &= \sum_i \sum_j P(y_j|x_i)P(x_i) \log P(y_j|x_i) - \sum_j \log P(y_j) \left(\sum_i P(x_i,y_j) \right) \\ &= \sum_i P(x_i) \sum_j P(y_j|x_i) \log P(y_j|x_i) - \sum_j P(y_j) \log P(y_j) \\ &= -H(y|x) + H(y) = H(y) - H(y|x) \\ &= H(x,y) - H(y|x) - H(x|y) = H(x) + H(y) - H(x,y) \end{aligned} \tag{B.202}$$

The last two qualities are due to Eq. (B.192) above. These relationships can be summarized in the following diagram:

B.2.3 Entropy Maximization

There may be the need to find an unknown distribution $p(x)$ of a random variable x based on the averages of a set of m observed independent experiment results $f_i(x)$:

$$\mathrm{E}[f_i(x)] = \int_{-\infty}^{\infty} p(x) f_i(x)\, dx = c_i \qquad (i = 1, \cdots, m) \tag{B.203}$$

Among all (infinite number of) possible density functions all satisfying these equations as well as $\int_{-\infty}^{\infty} p(x)\, dx = 1$, the one with the maximum entropy $H(p)$ is of the most interest as it allows maximum uncertainty and therefore imposes least amount of additional constraints. We can show that such a distribution has to take the following form:

$$p(x) = \exp\left(\lambda_0 + \sum_{i=1}^{m} \lambda_i f_i(x) \right) = A \exp\left(\sum_{i=1}^{m} \lambda_i f_i(x) \right) \tag{B.204}$$

where $\lambda_0, \cdots, \lambda_m$ and $A = e^{\lambda_0}$ are constants with values for $p(x)$ to satisfy Eq. (B.203).

To find this distribution $p(x)$ with maximal $H(p)$ and satisfying the constraints imposed by Eq. (B.203) as as well as $\int_{-\infty}^{\infty} p(x)\, dx = 1$ required of all probability density distributions, we need to solve the constrained optimization problem with the Lagrangian:

$$L(p) = \int_{-\infty}^{\infty} p(x) \log p(x)\, dx + \lambda_0 \left(\int_{-\infty}^{\infty} p(x) dx - 1 \right) + \sum_{i=1}^{m} \lambda_i \left(\int_{-\infty}^{\infty} p(x) f_i(x)\, dx - 1 \right) \tag{B.205}$$

We then take its derivative with respect to $p(x)$ corresponding to a specific value of x (i.e., x is treated as constant and the integral is ignored), and set it to zero:

$$\begin{aligned}\frac{d}{dp}L(p) &= \frac{d}{dp}(p\log p) + \lambda_0\frac{d}{dp}(p-1) + \sum_{i=1}^{m}\lambda_i\frac{d}{dp}(pf_i(x)-1) \\ &= \log p(x) + 1 - \lambda_0 - \sum_{i=1}^{m}\lambda_i f_i(x) = 0\end{aligned} \tag{B.206}$$

Solving this equation we get the optimal distribution in the form of Eq. (B.204):

$$p(x) = \exp\left(\lambda_0 - 1 + \sum_{i=1}^{m}\lambda_i f_i(x)\right) = A\exp\left(\sum_{i=1}^{m}\lambda_i f_i(x)\right) \tag{B.207}$$

where $A = \exp(\lambda_0 - 1)$.

Alternatively, the same result can also be obtained by comparing the entropies of an arbitrary density function $q(x)$, and another distribution $p(x)$ in the form of Eq. (B.204), both assumed to satisfy Eq. (B.203), i.e.,

$$\int p(x)f_i(x)\,dx = \int q(x)f_i(x)\,dx = c_i \quad (i = 1, \cdots, m) \tag{B.208}$$

We now find $H(q(x))$ and compare it with $H(p(x))$:

$$\begin{aligned}H(q) &= -\int q(x)\log q(x)\,dx = -\int q(x)\log\left(\frac{q(x)}{p(x)}p(x)\right)dx \\ &= -D_{KL}(q||p) - \int q(x)\log p(s)\,dx \le -\int q(x)\log p(s)\,dx \\ &= -\int q(x)\left(\lambda_0 + \sum_{i=1}^{m}\lambda_i f_i(x)\right)dx = -\int p(x)\left(\lambda_0 + \sum_{i=1}^{m}\lambda_i f_i(x)\right)dx \\ &= -\int p(x)\log p(x)\,dx = H(p)\end{aligned} \tag{B.209}$$

where we have used the fact that $D_{KL}(q||p) \ge 0$ and the result in Eq. (B.208). This result again indicates that the distribution with maximal $H(p)$ and satisfying Eq. (B.203) has to take the form in Eq. (B.204). The specific values of the $m+1$ constants $\lambda_0, \cdots, \lambda_m$ can be found by solving the $m+1$ equations including Eq. (B.203) and $\int_{-\infty}^{\infty} p(x)\,dx = 1$.

If the distribution $p(x)$ in question is further required to have zero mean and unit variance, i.e.,

$$\int p(x)x\,dx = 0, \qquad \int p(x)x^2\,dx = 1 \tag{B.210}$$

we can include two additional terms in Eq. (B.203) with $f_{m+1}(x) = x,\ f_{m+2} = x^2$ with $c_{m+1} = 0,\ c_{m+2} = 1$ so that

$$\int p(x)f_{m+1}dx = \int p(x)x\,dx = 0, \quad \int p(x)f_{m+2}dx = \int p(x)x^2\,dx = 1 \tag{B.211}$$

B.2.4 Kurtosis

The *kurtosis* of any random variable x with mean $\mu_x = \mathrm{E}[x]$ and variance $\sigma_x^2 = \mathrm{Var}(x) = \mathrm{E}[(x-\mu)^2]$ is defined as the fourth standardized moment:

$$\mathrm{Kurt}(x) = \mathrm{E}[z^4] = \mathrm{E}\left[\left(\frac{x-\mu_x}{\sigma_x}\right)^4\right] = \frac{\mathrm{E}[(x-\mu_x)^4]}{(\mathrm{E}[x-\mu_x])^4} \tag{B.212}$$

where $z = (x-\mu_x)/\sigma_x$ is the standardized version of x with zero-mean $\mu_z = \mathrm{E}[z] = 0$ and unity variance $\sigma_z^2 = \mathrm{Var}(z) = \mathrm{E}[(z-\mu_z)^2] = 1$. As the expectation of z^4, the kurtosis is relatively insensitive to values close to its zero mean inside the range $(-1,\ 1)$, but more sensitive to the outliers far away from its zero mean, i.e., kurtosis represents the "tailedness" of the density distribution of the variable x.

The *excess kurtosis* defined as $\mathrm{Kurt}(x) - 3$ is more widely used for comparison among different density distributions in terms of their tailedness. A given distribution is

- *Leptokurtic (super-Gaussian)* if $\mathrm{Kurt}(x) - 3 > 0$ (with heavier tails)
- *Mesokurtic* if $\mathrm{Kurt}(x) - 3 = 0$
- *Platykurtic (sub-Gaussian)* if $\mathrm{Kurt}(x) - 3 < 0$ (with lighter tails)

The figure below is a comparison of a set of distributions:

- Laplace distribution:

$$p(x) = \frac{1}{\sqrt{2}\sigma}\exp\left(-\sqrt{2}\left(\frac{|x-\mu|}{\sigma}\right)\right) \tag{B.213}$$

- Hyperbolic secant distribution:

$$p(x) = \frac{1}{2\sigma}\,\mathrm{sech}\left(\frac{\pi}{2}\left(\frac{x-\mu}{\sigma}\right)\right) \tag{B.214}$$

- Logistic distribution:

$$p(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2} = \frac{1}{4s}\,\mathrm{sech}^2\left(\frac{x-\mu}{2s}\right) \tag{B.215}$$

- Normal distribution:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{B.216}$$

- Wigner semicircle distribution:

$$p(x) = \begin{cases} \frac{2}{\pi R^2}\sqrt{R^2-x^2} & -R \le x \le R \\ 0 & \text{otherwise} \end{cases} \tag{B.217}$$

- uniform distribution:

$$p(x) = \begin{cases} 1/(b-a) & a \le x \le b \\ 0 & \text{otherwise} \end{cases} \tag{B.218}$$

	Laplace	Hyperbolic Secant	Logistic*	Normal	Semicircle**	Uniform
Mean	μ	μ	μ	μ	0	$(a+b)/2$
Variance	σ^2	σ^2	σ^2	σ^2	σ^2	$(b-a)^2/1$
Entropy	$\ln(\sqrt{2}\sigma e)$	1.166	$\ln(\sqrt{3}\sigma/\pi)$	$\ln(2\pi e\sigma^2)$	$\ln(2\pi\sigma) - 1/2$	$\ln(b-a)$
E-Kurtosis	3	2	1.2	0	-1	-1.2

(B.219)

* $\sigma = s\pi/\sqrt{3}$, ** $\sigma = R/2$.

B.3 Bayesian Inference

B.3.1 Frequentist versus Bayesian Statistics

In machine learning it is commonly the case that all observed data D are interpreted probabilistically, i.e., all samples in dataset $\mathbf{X}$ are assumed to be random vectors with certain probability distributions $p(\mathbf{x})$. While building a hypothetical model parameterized by θ to fit the data, we can used either of two different approaches, the *frequentist approach* that treats all samples in data D as random variables and assigns probabilities to them, and the *Bayesian approach* that also treats the model parameters θ as random variables and assigns probabilities to them as well as to data D. More specifically, the Bayesian approach is based on a prior probability $p(\theta)$ of the parameter before observing any data, which is then improved to become the posterior probability $p(\theta|D)$ after observing the data D.

Consider, as a simple example, the estimation of the ratio θ of red balls in a box containing a large number of balls based on the observed data D: taking n random samples from the box (one at a time with replacement) and finding $k \le n$ of them red.

- **The frequentist approach:**

  The probability of finding k red balls in n samples is $P(D) = P(n,k) = \theta^k(1-\theta)^{n-k}$ (Bernoulli distribution), which, as the probability of an event that has already happened, should be the maximum among all possible values of θ, i.e., it should satisfy $d\,P(D)/d\theta = 0$ or equivalently

$$\frac{d}{d\theta}\ln P(D) = \frac{d}{d\theta}[k\ln\theta + (n-k)\ln(1-\theta)] = \frac{k}{\theta} - \frac{n-k}{1-\theta} = 0 \tag{B.220}$$

  Solving this equation we get the frequentist model parameter $\theta = k/n$.

- **The Bayesian approach:**

  The prior probability of θ has a uniform distribution $P(\theta) = 1$, as all values $\theta \in [0, 1]$ are equally likely without knowledge from observing any data ($P(\theta) = 0$ for any $\theta < 0$ or $\theta > 1$). The likelihood of θ is the probability of observing data D (in terms of n and k) conditioned on a given θ, i.e., $p(D|\theta) = p(n, k|\theta) = \theta^k(1-\theta)^{n-k}$. Now we can further find the posterior:

$$p(\theta|D) = \frac{p(D|\theta)\, p(\theta)}{p(D)} = \frac{p(D|\theta)\, p(\theta)}{\int_0^1 p(D|\theta)p(\theta)d\theta} \tag{B.221}$$

  Substituting into this expression $p(\theta)$, $p(D|\theta)$ and the denominator (a beta function)

$$\int_0^1 p(D|\theta)p(\theta)d\theta = \int_0^1 \theta^k(1-\theta)^{n-k}d\theta = \frac{k!(n-k)!}{(n+1)!} \tag{B.222}$$

  we get

$$p(\theta|D) = \frac{(n+1)!}{k!(n-k)!}\theta^k(1-\theta)^{n-k} \tag{B.223}$$

  based on which we further find the expectation:

$$\begin{aligned} E[\theta] &= \int_0^1 \theta\, p(\theta|D)d\theta = \frac{(n+1)!}{k!(n-k)!}\int_0^1 \theta^{k+1}(1-\theta)^{n-k} \\ &= \frac{(n+1)!}{k!(n-k)!}\frac{(k+1)!(n-k)!}{(n+2)!} = \frac{k+1}{n+2} \end{aligned} \tag{B.224}$$

  This is called Laplace's rule of succession.

For example, when $n = 2$ there exist three possible outcomes:

k	Frequentist θ	Bayesian θ
0	0	1/4
1	1/2	1/2
2	1	3/4

(B.225)

It is obvious that for such a small sample set, the frequentist results are more absolute ($k = 0$ and $k = 2$) and less reliable, while the Bayesian results are less absolute and therefore more reliable. But these two types of models tend to be the same when n approaches infinity.

As an example with a very large n, consider the sunrise problem for finding the probability that the sun will rise tomorrow, given that it has risen every day for the past 5,000 years, i.e., $k = n = 5000 \times 365.25$. This probability is $\theta = k/n = 1$ according to the frequentist model, but $\theta = (k+1)/(n+2) = 0.99999945$ according to Eq. (B.224) by Laplace.

B.3.2 Bayesian Inference

The method of *Bayesian inference* based on Bayes' theorem given in Eq. (B.12) is widely used in many machine learning algorithms. In general, Bayesian inference deduces a hypothesis H, a model of some observed dataset D in terms of its probability distribution. Bayesian inference can be an iterative process that updates the model parameters as more data becomes available. Bayesian inference can be formulated in words as

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} \tag{B.226}$$

or symbolically

$$p(H|D) = \frac{p(D|H)\,p(H)}{p(D)} \propto p(D|H)\,p(H) \tag{B.227}$$

where $p(H)$ is the *prior probability* of the hypothesis H based on some prior knowledge without observation of any data, $p(H|D)$ is the *posterior probability* of H based on the observation of some data D, $p(D|H)$ is the *likelihood* of H, proportional to the probability of observing D given H, and $p(D)$ is the probability of observing data D, which can be considered as a normalizing factor and is independent of H in question, and can therefore be dropped. Specifically, data D and hypothesis H take the following forms:

- Data D: a collection of data points in a d-dimensional *feature space*, represented in a data matrix $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ containing a set of N independent samples of a vector $\mathbf{x} = [x_1, \cdots, x_d]^T$, randomly and uniformly drown from a population with certain probability density distribution. Such samples are therefore *independent and identically distributed (i.i.d.)*. In supervised learning, the given dataset D, now called *training set*, also include a vector $\mathbf{y} = [y_1, \cdots, y_N]^T$, of which component y_i is a label for the corresponding $\mathbf{x}_i$ in $\mathbf{X}$.
- Hypothesis H: a specific probabilistic model $p(\mathbf{x}|\boldsymbol{\theta})$ for the dataset, parameterized by $\theta_1, \cdots, \theta_M$ represented as a vector $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_M]^T$, to be estimated based on the observed dataset $\mathbf{X}$.

Bayes inference can now be formulated as a process for estimating the model parameter $\boldsymbol{\theta}$ based on the given dataset $\mathbf{X}$ (possibly also $\mathbf{y}$), and Eq. (B.227) can be written as:

$$p(\boldsymbol{\theta}|\mathbf{X}) = \frac{p(\mathbf{X}|\boldsymbol{\theta})\,p(\boldsymbol{\theta})}{p(\mathbf{X})} = \frac{p(\mathbf{X}|\boldsymbol{\theta})\,p(\boldsymbol{\theta})}{\sum_\theta p(\mathbf{X}|\boldsymbol{\theta})\,p(\boldsymbol{\theta})} \propto p(\mathbf{X}|\boldsymbol{\theta})\,p(\boldsymbol{\theta}) \tag{B.228}$$

where

- $p(\boldsymbol{\theta})$: The *a priori* probability, or the simply *prior*, of parameter $\boldsymbol{\theta}$ based on any prior knowledge before observing data $\mathbf{X}$.
- $L(\boldsymbol{\theta}|\mathbf{X}) \propto p(\mathbf{X}|\boldsymbol{\theta})$: The *likelihood* of the model parameter $\boldsymbol{\theta}$ (hypothesis), given the observed data $\mathbf{X}$, measured in terms of the probability of observing $\mathbf{X}$

conditioned on a value of $\boldsymbol{\theta}$. In maximum likelihood estimation, we need to find the optimal $\boldsymbol{\theta}$ that maximizes the likelihood, $L(\boldsymbol{\theta}|\mathbf{X})$ does not need to be normalized as $p(\mathbf{X}|\boldsymbol{\theta})$ If the data samples in $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ are independently drawn from the same pdf, i.e., they are i.i.d. samples, then

$$p(\mathbf{X}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p(\mathbf{x}_n|\boldsymbol{\theta}) \tag{B.229}$$

- $p(\mathbf{X}) = \sum_\theta p(\mathbf{X}|\theta)p(\theta)$: The distribution of data D while the parameter θ is marginalized. It can also be considered as a normalization factor to guarantee that the resulting posterior is a probability satisfying $\sum_\theta p(\boldsymbol{\theta}|\mathbf{X}) = 1$. As $p(\mathbf{X})$ is a constant independent of $\boldsymbol{\theta}$ in question, it is can be dropped for convenience when maximizing the posterior.
- $p(\theta|\mathbf{X})$: The *a posteriori* probability, or simply the *posterior*, of the parameter $\boldsymbol{\theta}$, after observing data $\mathbf{X}$.

There are two methods typically used for the estimation of the parameters in $\boldsymbol{\theta}$ based on the observed data $\mathbf{X}$:

- *Maximum Likelihood Estimation (MLE)* seeks to find $\boldsymbol{\theta}$ that maximizes the likelihood:

$$\boldsymbol{\theta}_{MLE} = \arg\max_{\theta} \; p(\mathbf{X}|\boldsymbol{\theta}) \tag{B.230}$$

- *Maximum A Posteriori (MAP)* seeks to find $\boldsymbol{\theta}$ that maximizes the posterior:

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\theta} \; p(\boldsymbol{\theta}|\mathbf{X}) \propto p(\mathbf{X}|\boldsymbol{\theta})\, p(\boldsymbol{\theta}) \tag{B.231}$$

The difference between the two methods is whether certain prior knowledge is available regarding the distribution $p(\boldsymbol{\theta})$ of the parameters before observing any data. If no such prior knowledge is available, then we can only assume all possible values of the parameters are equally likely, i.e., the prior probability $p(\boldsymbol{\theta}) =$ constant is a uniform distribution, and the two methods will produce the same result. On the other hand, if we have some prior knowledge that the parameters obey certain non-uniform distribution $p(\boldsymbol{\theta})$, e.g., they are normally distributed with certain mean and covariance, then the MAP method may produce more accurate estimation.

When we maximize either the likelihood or the posterior as a function of $\boldsymbol{\theta}$, we can equivalently maximize its logarithm (a monotonic function) for mathematical convenience, so that the two methods above can be expressed as

- MLE:

$$\begin{aligned}\boldsymbol{\theta}_{MLE} &= \arg\max_{\theta} \; \log \; p(\mathbf{X}|\boldsymbol{\theta}) = \arg\max_{\theta} \; \log \left(\prod_{n=1}^{N} p(\mathbf{x}_n|\boldsymbol{\theta}) \right) \\ &= \arg\max_{\theta} \left(\sum_{n=1}^{N} \log \; p(\mathbf{x}_n|\boldsymbol{\theta}) \right)\end{aligned} \tag{B.232}$$

- MAP:

$$\begin{aligned}\boldsymbol{\theta}_{MAP} &= \arg\max_{\theta} \log\ (p(\mathbf{X}|\boldsymbol{\theta})\, p(\boldsymbol{\theta})) \\ &= \arg\max_{\theta} \left(\sum_{n=1}^{N} \log\ p(\mathbf{x}_n|\boldsymbol{\theta}) + \log\, p(\boldsymbol{\theta}) \right) \end{aligned} \tag{B.233}$$

Both MLE and MAP seek to find some specific values for the parameters in $\boldsymbol{\theta}$ that maximize certain function of the parameters, such as the likelihood or the posterior of $\boldsymbol{\theta}$, or their logarithm functions. In certain situations, we can instead find the parameter $\boldsymbol{\theta}$ that maximizes the expectation of the function to be maximized, denoted by $Q(\boldsymbol{\theta})$, by the method called *expectation-maximization (EM)*, in an iteration of the following two steps:

- The E-step: find the expectation $Q(\boldsymbol{\theta})$ based on the previous parameter values $\boldsymbol{\theta}_n$;
- The M-step: find the updated parameter values $\boldsymbol{\theta}_{n+1}$ that maximize the expectation $Q(\boldsymbol{\theta})$.

B.3.3 Maximum Likelihood Estimation of Model Parameters

Given an observed dataset $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$ containing N independent and identically distributed (i.i.d.) sample data points, we can build a statistical model $p(\mathbf{x}|\boldsymbol{\theta})$, of which the parameter $\boldsymbol{\theta}$ containing a set of model parameters is to be estimated based on the observed dataset $\mathbf{X}$ by method such as *maximum likelihood estimation (MLE)*.

Specifically, we let $\boldsymbol{\theta}'$ be the current estimation of the true model parameter $\boldsymbol{\theta}$, and find its likelihood and log-likelihood:

$$L(\boldsymbol{\theta}'|\mathbf{x}) \propto p(\mathbf{x}|\boldsymbol{\theta}'), \qquad l(\boldsymbol{\theta}'|\mathbf{x}) = \log p(\mathbf{x}|\boldsymbol{\theta}') \tag{B.234}$$

We further consider the expectation of the log-likelihood with respect to the true distribution $p(\mathbf{x}|\boldsymbol{\theta})$ is

$$\begin{aligned} E_\theta[l(\boldsymbol{\theta}'|\mathbf{x})] &= \int p(\mathbf{x}|\boldsymbol{\theta}) l(\boldsymbol{\theta}'|\mathbf{x}) d\mathbf{x} = \int p(\mathbf{x}|\boldsymbol{\theta}) \log p(\mathbf{x}|\boldsymbol{\theta}') d\mathbf{x} \\ &= \int p(\mathbf{x}|\boldsymbol{\theta}) \log \left(p(\mathbf{x}|\boldsymbol{\theta}) \frac{p(\mathbf{x}|\boldsymbol{\theta}')}{p(\mathbf{x}|\boldsymbol{\theta})} \right) d\mathbf{x} \\ &= \int p(\mathbf{x}|\boldsymbol{\theta}) \log p(\mathbf{x}|\boldsymbol{\theta}) d\mathbf{x} - \int p(\mathbf{x}|\boldsymbol{\theta}) \log \frac{p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})'} d\mathbf{x} \\ &= -H(p(\mathbf{x}|\boldsymbol{\theta})) - D_{KL}(p(\mathbf{x}|\boldsymbol{\theta}))||p(\mathbf{x}|\boldsymbol{\theta}')) \end{aligned} \tag{B.235}$$

where $H(p(\mathbf{x}|\boldsymbol{\theta}))$ is the entropy of $p(\mathbf{x}|\boldsymbol{\theta}))$, which is independent of the estimated parameter $\boldsymbol{\theta})'$, and $D_{KL}(p(\mathbf{x}|\boldsymbol{\theta}))||p(\mathbf{x}|\boldsymbol{\theta}'))$ is the KL-divergence between $p(\mathbf{x}|\boldsymbol{\theta}))$ and $p(\mathbf{x}|\boldsymbol{\theta}'))$, which is zero when the estimated parameter becomes the same as the true parameter $\boldsymbol{\theta}' = \boldsymbol{\theta}$.

Now we see that the optimal model parameter that maximizes the likelihood is also the one that minimizes the KL-divergence:

$$\boldsymbol{\theta}_{MLE} = \arg\max_{\theta'} \; L(\boldsymbol{\theta}'|\mathbf{x}) = \arg\min_{\theta'} \; D_{KL}(p(\mathbf{x}|\boldsymbol{\theta}))||p(\mathbf{x}|\boldsymbol{\theta}')) \tag{B.236}$$

Consider the simple example of the MLE method applied to estimate the two parameters in $\boldsymbol{\theta} = \{\mathbf{m}, \boldsymbol{\Sigma}\}$ of the multivariate Gaussian distribution

$$\mathcal{N}(\mathbf{x}, \mathbf{m}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2} \, |\boldsymbol{\Sigma}|^{1/2}} \; \exp\left[-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\mathbf{m})\right] \tag{B.237}$$

Given an observed dataset containing N i.i.d. data samples $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$, we can get the likelihood function

$$L(\boldsymbol{\theta}|\mathbf{X}) = L(\mathbf{m}, \boldsymbol{\Sigma}|\mathbf{X}) = p(\mathbf{X}|\mathbf{m}, \boldsymbol{\Sigma}) = \prod_{i=1}^{N} \mathcal{N}(\mathbf{x}_i|\mathbf{m}, \boldsymbol{\Sigma}) \tag{B.238}$$

and the log-likelihood function:

$$\begin{aligned} l(\mathbf{m}, \boldsymbol{\Sigma}|\mathbf{X}) = \log \; L(\mathbf{m}, \boldsymbol{\Sigma}|\mathbf{X}) &= \log\left[\prod_{i=1}^{N} \mathcal{N}(\mathbf{x}_i|\mathbf{m}, \boldsymbol{\Sigma})\right] = \sum_{i=1}^{N} \log \mathcal{N}(\mathbf{x}_i|\mathbf{m}, \boldsymbol{\Sigma}) \\ &= \sum_{i=1}^{N}\left[-\frac{d}{2}\log 2\pi - \frac{1}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x}_i-\mathbf{m})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\mathbf{m})\right] \end{aligned} \tag{B.239}$$

To find the optimal parameters $\mathbf{m}$ and $\boldsymbol{\Sigma}$ that maximize this log-likelihood, we set its derivative with respect to $\mathbf{m}$ and $\boldsymbol{\Sigma}$ to zero and solve the resulting equations:

- Find $\mathbf{m}$

$$\begin{aligned} \frac{\partial}{\partial \mathbf{m}} l(\mathbf{m}, \boldsymbol{\Sigma}|\mathbf{X}) &= -\frac{1}{2}\sum_{i=1}^{N} \frac{\partial}{\partial \mathbf{m}} \left((\mathbf{x}_i-\mathbf{m})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\mathbf{m})\right) \\ &= -\frac{1}{2}\sum_{i=1}^{N} \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\mathbf{m}) = \mathbf{0} \end{aligned} \tag{B.240}$$

  Pre-multiplying $-2\boldsymbol{\Sigma}$ on both sides, we get

$$\sum_{i=1}^{N}(\mathbf{x}_i-\mathbf{m}) = \mathbf{0} \tag{B.241}$$

  solving which we get the optimal estimation of $\mathbf{m}$:

$$\hat{\mathbf{m}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i \tag{B.242}$$

- Find $\boldsymbol{\Sigma}$

$$\frac{\partial}{\partial \boldsymbol{\Sigma}} l(\mathbf{m}, \boldsymbol{\Sigma}|\mathbf{X}) = -\frac{N}{2}\frac{\partial}{\partial \boldsymbol{\Sigma}} \log|\boldsymbol{\Sigma}| - \frac{1}{2}\sum_{i=1}^{N} \frac{\partial}{\partial \boldsymbol{\Sigma}} \left((\mathbf{x}_i - \mathbf{m})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \mathbf{m})\right)$$
$$= -\frac{N}{2}\boldsymbol{\Sigma}^{-1} + \frac{1}{2}\sum_{i=1}^{N} \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \mathbf{m})(\mathbf{x}_i - \mathbf{m})^T \boldsymbol{\Sigma}^{-1} = 0 \quad \text{(B.243)}$$

Pre and post multiplying $\boldsymbol{\Sigma}$ on both sides and solving the resulting equation we get the optimal estimation of $\boldsymbol{\Sigma}$:

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \mathbf{m})(\mathbf{x}_i - \mathbf{m})^T \quad \text{(B.244)}$$

Here we have used the following facts:

$$\frac{d}{d\mathbf{A}} \log|\mathbf{A}| = (\mathbf{A}^{-1})^T \quad \text{(B.245)}$$

$$\frac{d}{d\mathbf{A}} \left(\mathbf{a}^T \mathbf{A}^{-1} \mathbf{b}\right) = -(\mathbf{A}^{-1})^T \mathbf{a}\mathbf{b}^T (\mathbf{A}^{-1})^T \quad \text{(B.246)}$$

Note that the rank of $\hat{\boldsymbol{\Sigma}}$ in Eq. (B.244) is $N-1$, representing its degrees of freedom due to the N samples, assumed to be independent, and the extra constraint in Eq. (B.242). When $d > N-1$, the matrix $\hat{\boldsymbol{\Sigma}}$ does not have a full rank and is therefore non-invertible.

For more details regarding derivatives of a scalar function with respect to vector and matrix variables, see *Matrix Cookbook* here: `https://www.math.uwaterloo.ca/~h`

B.3.4 Natural gradient

The gradient method is commonly used in optimization to minimize or maximize an objective function $J(\mathbf{x})$ in Euclidean space based on the following iteration:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}^* \quad \text{(B.247)}$$

where $\Delta\mathbf{x}^*$ is the optimal increment among all those $\Delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n$ within a small neighborhood of the current $\mathbf{x}_n$:

$$\Delta\mathbf{x}^* = \underset{||\Delta\mathbf{x}||<C}{\arg\max}\, J(\mathbf{x} + \Delta\mathbf{x}) \quad \text{(B.248)}$$

The solution of this constrained optimization problem can be found by maximizing the following Lagrangian function

$$L(\Delta\mathbf{x}) = L(\mathbf{x} + \Delta\mathbf{x}) + \mu(||\Delta\mathbf{x}||^2 - C) \quad \text{(B.249)}$$

The optimal $\Delta\mathbf{x}^*$ can be found by setting the derivative of $L(\Delta\mathbf{x})$ to zero

$$\frac{d}{d\,\Delta\mathbf{x}} L(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{g}_L(\mathbf{x}) + \frac{\mu}{2}\Delta\mathbf{x} = 0 \quad \text{(B.250)}$$

and solving the resulting equation to get

$$\Delta \mathbf{x}^* = -\frac{2}{\mu}\mathbf{g}_J(\mathbf{x}) \tag{B.251}$$

Now the optimal $\mathbf{x}^*$ can be found iteratively

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta \mathbf{x}_n = \mathbf{x}_n - \lambda \mathbf{g}_J(\mathbf{x}_n) \tag{B.252}$$

where the step size $\lambda = 2/\mu$ can be adjusted as needed.

However, in some cases the function to be optimized is not in the Euclidean space spanned by a set of orthonormal basis vector in which an inner product is defined and the Euclidean distance is used to measure the difference between two points in the space, but in a *Riemannian manifold* (a smooth manifold locally similar to a Euclidean space), in which a small neighboring area of a point can be approximated by its tangent space spanned by a set of vectors $\{\mathbf{e}_1, \cdots, \mathbf{e}_n\}$ not necessarily orthogonal to each other, and the squared length of an infinitesimal arc $d\mathbf{x} = [dx_1, \cdots, dx_d]^T$ on the manifold can be expressed in quadratic form as:

$$\begin{aligned} ||d\mathbf{x}||^2 &= \left(\sum_{i=1}^{n} dx_i \mathbf{e}_i\right)^T \left(\sum_{j=1}^{n} dx_j \mathbf{e}_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} dx_i dx_j \mathbf{e}_i^T \mathbf{e}_j \\ &= d\mathbf{x}^T \mathbf{G}\, d\mathbf{x} \end{aligned} \tag{B.253}$$

Here $\mathbf{G}$, called the *metric tensor*, is a positive-definite matrix based on the inner products of the basis vectors in the tangent space:

$$\mathbf{G} = \begin{bmatrix} \mathbf{e}_1^T\mathbf{e}_1 & \cdots & \mathbf{e}_1^T\mathbf{e}_n \\ \vdots & \ddots & \vdots \\ \mathbf{e}_n^T\mathbf{e}_1 & \cdots & \mathbf{e}_n^T\mathbf{e}_n \end{bmatrix} \tag{B.254}$$

In the special case when the Riemannian space is a Euclidean space spanned by orthonormal basis vectors satisfying $\mathbf{e}_i^T\mathbf{e}_j = \delta_{ij}$, we have $\mathbf{G} = \mathbf{I}$, and the squared arc length above becomes the conventional Euclidean squared norm based on inner product: $||d\mathbf{x}||^2 = d\mathbf{x}^T d\mathbf{x} = \sum_i dx_i^2$.

To minimize an objective function $J(\mathbf{x})$ in a Riemannian manifold iteratively, we can still find the optimal increment step $\Delta\mathbf{x}$ as given in Eq. (B.248), that maximizes the Lagrangian in Eq. (B.249), but now the squared norm is given in Eq. (B.253):

$$L(\Delta\mathbf{x}) = L(\mathbf{x} + \Delta\mathbf{x}) + \mu(\Delta\mathbf{x}^T\mathbf{G}\Delta\mathbf{x} - C) \tag{B.255}$$

Now we solve the equation

$$\begin{aligned} \frac{d}{d\,\Delta\mathbf{x}} L(\mathbf{x} + \Delta\mathbf{x}) &= \frac{d}{d\,\Delta\mathbf{x}} L(\mathbf{x} + \Delta\mathbf{x}) + \mu \frac{d}{d\,\Delta\mathbf{x}} \left(\Delta\mathbf{x}^T\mathbf{G}\Delta\mathbf{x} - C\right) \\ &= \mathbf{g}_L(\mathbf{x}) + \frac{\mu}{2}\mathbf{G}\Delta\mathbf{x} \end{aligned} \tag{B.256}$$

to get the optimal increment:

$$\Delta\mathbf{x}^* = -\frac{\mu}{2}\mathbf{G}^{-1}\mathbf{g}_L(\mathbf{x}) = -\frac{\mu}{2}\tilde{\mathbf{g}}_L(\mathbf{x}) \tag{B.257}$$

where $\tilde{\mathbf{g}}_L(\mathbf{x}) = \mathbf{G}^{-1}\mathbf{g}_L(\mathbf{x})$ is the *natural gradient*, and the iteration becomes

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_n = \mathbf{x}_n - \delta\tilde{\mathbf{g}}_L(\mathbf{x}) \tag{B.258}$$

B.3.5 Fisher Information and Natural Gradient

The *Fisher information* measures the amount of information carried by an observed random $\mathbf{x}$ about the unknown parameter $\boldsymbol{\theta}$ of a model distribution $p(\mathbf{x}|\boldsymbol{\theta})$ of $\mathbf{x}$.

The likelihood function of the parameter $\boldsymbol{\theta}$ for a model $p(\mathbf{x}|\boldsymbol{\theta})$ given observed data $\mathbf{x}$ is

$$L(\boldsymbol{\theta}|\mathbf{x}) \propto p(\mathbf{x}|\boldsymbol{\theta}) \tag{B.259}$$

and the log-likelihood is

$$l(\boldsymbol{\theta}|\mathbf{x}) = \log\, L(\boldsymbol{\theta}|\mathbf{x}) = \log p(\mathbf{x}|\boldsymbol{\theta}) \tag{B.260}$$

The gradient of the log-likelihood is

$$\mathbf{g}_l(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta}) = \frac{d\, p(\mathbf{x}|\boldsymbol{\theta})/d\boldsymbol{\theta}}{p(\mathbf{x}|\boldsymbol{\theta})} = \frac{\mathbf{g}_p(\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})} \tag{B.261}$$

where $\mathbf{g}_p(\boldsymbol{\theta}) = d\, p(\mathbf{x}|\boldsymbol{\theta})/d\boldsymbol{\theta}$ is the gradient of $p(\mathbf{x}|\boldsymbol{\theta})$. The expectation of $\mathbf{g}_l$ is zero:

$$\begin{aligned} E_{\boldsymbol{\theta}}[\mathbf{g}_l(\boldsymbol{\theta})] &= \int \left(\frac{\mathbf{g}_p(\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right) p(\mathbf{x}|\boldsymbol{\theta}) d\mathbf{x} = \int \mathbf{g}_p(\boldsymbol{\theta}) d\mathbf{x} \\ &= \int \frac{d}{d\boldsymbol{\theta}}\, p(\mathbf{x}|\boldsymbol{\theta}) d\mathbf{x} = \frac{d}{d\boldsymbol{\theta}} \int p(\mathbf{x}|\boldsymbol{\theta}) d\mathbf{x} = \frac{d}{d\boldsymbol{\theta}}\, 1 = \mathbf{0} \end{aligned} \tag{B.262}$$

and the covariance of $\mathbf{g}_l$ is defined as the *Fisher information*:

$$\mathbf{F}(\boldsymbol{\theta}) = E_{\boldsymbol{\theta}}[\mathbf{g}_l(\boldsymbol{\theta})\mathbf{g}_l^T(\boldsymbol{\theta})] \tag{B.263}$$

which can be estimated based on an observed dataset $\mathbf{x} = [\mathbf{x}_1, \cdots, \mathbf{x}_N]$:

$$\mathbf{F}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \mathbf{g}_l(\boldsymbol{\theta})\mathbf{g}_l^T(\boldsymbol{\theta}) \tag{B.264}$$

The Hessian of the log-likelihood $l(\boldsymbol{\theta}|\mathbf{x})$ is:

$$\begin{aligned} \mathbf{H}_l(\boldsymbol{\theta}) &= \frac{d^2}{d\boldsymbol{\theta}^2} l(\boldsymbol{\theta}|\mathbf{x}) = \frac{d}{d\boldsymbol{\theta}} \mathbf{g}_l(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}} \left[\frac{\mathbf{g}_p(\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right] \\ &= \frac{\mathbf{H}_p(\boldsymbol{\theta}) p(\mathbf{x}|\boldsymbol{\theta}) - \mathbf{g}_p(\boldsymbol{\theta})\mathbf{g}_p^T(\boldsymbol{\theta})}{p^2(\mathbf{x}|\boldsymbol{\theta})} = \frac{\mathbf{H}_p(\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})} - \mathbf{g}_l(\boldsymbol{\theta})\mathbf{g}_l^T(\boldsymbol{\theta}) \end{aligned} \tag{B.265}$$

The expectation of $\mathbf{H}_l$ is

$$E_{\boldsymbol{\theta}}[\mathbf{H}_l(\boldsymbol{\theta})] = E_{\boldsymbol{\theta}}\left[\frac{\mathbf{H}_p(\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right] - E_{\boldsymbol{\theta}}\left[\mathbf{g}_l(\boldsymbol{\theta})\mathbf{g}_l^T(\boldsymbol{\theta})\right] \tag{B.266}$$

The first term is

$$E_{\boldsymbol{\theta}}\left[\frac{\mathbf{H}_p(\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right] = \int \left[\frac{\mathbf{H}_p(\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right] p(\mathbf{x}|\boldsymbol{\theta})d\mathbf{x} = \int \mathbf{H}_p(\boldsymbol{\theta})d\mathbf{x} = \int \frac{d^2}{d\boldsymbol{\theta}^2} p(\mathbf{x}|\boldsymbol{\theta})d\mathbf{x}$$
$$= \frac{d^2}{d\boldsymbol{\theta}^2}\int p(\mathbf{x}|\boldsymbol{\theta})d\mathbf{x} = \frac{d^2}{d\boldsymbol{\theta}^2}\, 1 = \mathbf{0} \tag{B.267}$$

and the second term is the Fisher information defined in Eq. (B.263), and we have:

$$\mathbf{F}(\boldsymbol{\theta}) = -E_{\boldsymbol{\theta}}[\mathbf{H}_l(\boldsymbol{\theta})] = E_{\boldsymbol{\theta}}\left[\mathbf{g}_l(\boldsymbol{\theta})\mathbf{g}_l^T(\boldsymbol{\theta})\right] \tag{B.268}$$

of which the component in the ith row and jth column is expected product of the ith and jth components of $\mathbf{g}_l(\boldsymbol{\theta})$:

$$F_{ij} = \mathrm{E}[g_i\ g_j] = \mathrm{E}\left[\frac{\partial \log p(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_i}\frac{\partial \log p(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_j}\right] \tag{B.269}$$

We note that $g_i = \partial \log p(\mathbf{x}|\boldsymbol{\theta})/\partial\theta_i$ represents how much variation in the pdf $p(\mathbf{x}|\boldsymbol{\theta})$ of $\mathbf{x}$ will be caused by parameter θ_i, or how much information about θ_i is carried by $\mathbf{x}$. If in the extreme case when $g_i = 0$, then $\mathbf{x}$ carries no information about θ_i as it is irrelevant to θ_i. Therefore the Fisher information containing all $F_{ij} = \mathrm{E}[g_i\ g_j]$ represents the amount of information about $\boldsymbol{\theta}$ carried by $\mathbf{x}$.

While building a statistical model $p(\mathbf{x}|\boldsymbol{\theta})$ of the observed dataset, we need to approximate the unknown parameter $\boldsymbol{\theta}$ by the estimated $\boldsymbol{\theta}'$ based on the samples of variable $\mathbf{x}$, so that the KL-divergence between $p(\mathbf{x}|\boldsymbol{\theta}')$ and the true $p(\mathbf{x}|\boldsymbol{\theta})$ (Eq. (B.198)) is minimized:

$$D_{KL}(p(\mathbf{x}|\boldsymbol{\theta})||p(\mathbf{x}|\boldsymbol{\theta}')) = E_{\boldsymbol{\theta}}[\log p(\mathbf{x}|\boldsymbol{\theta}) - \log p(\mathbf{x}|\boldsymbol{\theta}')] \tag{B.270}$$

Consider its first and second order derivatives with respect to $\boldsymbol{\theta}'$:

- The gradient:

$$\mathbf{g}_{KL}(\boldsymbol{\theta}') = E_{\boldsymbol{\theta}}\left[\frac{d}{d\boldsymbol{\theta}'}[\log p(\mathbf{x}|\boldsymbol{\theta}) - \log p(\mathbf{x}|\boldsymbol{\theta}')]\right]$$
$$= -E_{\boldsymbol{\theta}}\left[\frac{d}{d\boldsymbol{\theta}'}\log p(\mathbf{x}|\boldsymbol{\theta}')\right] = -E_{\boldsymbol{\theta}}[\mathbf{g}_l(\boldsymbol{\theta}')] \tag{B.271}$$

- The Hessian:

$$\mathbf{H}_{KL}(\boldsymbol{\theta}') = -E_{\boldsymbol{\theta}}\left[\frac{d}{d\boldsymbol{\theta}'}\mathbf{g}_l(\boldsymbol{\theta}')\right] = -E_{\boldsymbol{\theta}}[\mathbf{H}_l(\boldsymbol{\theta}')] \tag{B.272}$$

Evaluating both $\mathbf{g}_{KL}(\boldsymbol{\theta}')$ and $\mathbf{H}_{KL}(\boldsymbol{\theta}')$ at $\boldsymbol{\theta}' = \boldsymbol{\theta}$, we get (Eqs. (B.262) and (B.268)):

$$\mathbf{g}_{KL}(\boldsymbol{\theta}) = -E_{\boldsymbol{\theta}}[\mathbf{g}_l(\boldsymbol{\theta})] = \mathbf{0}, \qquad \text{and} \qquad \mathbf{H}_{KL}(\boldsymbol{\theta}) = \mathbf{F}(\boldsymbol{\theta}) \tag{B.273}$$

Now the KL-divergence above can be approximated by the first three terms of

its Taylor expansion in the neighborhood of $\boldsymbol{\theta}$ in terms of $\Delta\boldsymbol{\theta} = \boldsymbol{\theta}' - \boldsymbol{\theta}$:

$$D_{KL}(p(\mathbf{x}|\boldsymbol{\theta})||p(\mathbf{x}|\boldsymbol{\theta}')) \approx D_{KL}(p(\mathbf{x}|\boldsymbol{\theta})||p(\mathbf{x}|\boldsymbol{\theta})) + \mathbf{g}_{KL}^T(\boldsymbol{\theta})\Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^T\mathbf{H}_{KL}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} = \frac{1}{2}\Delta\boldsymbol{\theta}^T\mathbf{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} \quad \text{(B.274)}$$

Comparing this with Eq. (B.253), we see that we can treat each component $g_i = \partial \log p(\mathbf{x}|\boldsymbol{\theta})/\partial\theta_j$ of $\mathbf{g}_l(\boldsymbol{\theta})$ as a basis vector in the Riemanniian space for all distributions $p(\mathbf{x}|\boldsymbol{\theta})$, then the KL-divergence above becomes the squared distance between the two distributions:

$$D_{KL}(p(\mathbf{x}|\boldsymbol{\theta})||p(\mathbf{x}|\boldsymbol{\theta}')) \approx \frac{1}{2}\Delta\boldsymbol{\theta}^T\mathbf{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} = \frac{1}{2}\sum_i\sum_j \Delta\theta_i\Delta\theta_j g_i g_j \quad \text{(B.275)}$$

Now the optimal model parameter $\boldsymbol{\theta}$ that maximizes the likelihood or equivalently minimizes the KL-divergence (Eq. (B.236)) can be found iteratively by the natural gradient method (Eq (B.258)):

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \Delta\boldsymbol{\theta}_n = \boldsymbol{\theta}_n + \delta\tilde{\mathbf{g}}_L(\boldsymbol{\theta}_n) \quad \text{(B.276)}$$

B.3.6 Bias-Variance Trade-off

The *mean squared error* as defined below is a common way to measure the error of the estimation of some parameter, or, more generally, the performance of a supervised learning algorithm (for either regression or classification). Let $\hat{\theta}$ be an estimator of the parameter θ of a data model. Then *mean squared error (MSE)* of this estimator is

$$\begin{aligned}\text{MSE}(\hat{\theta}) &= \text{E}[(\hat{\theta} - \theta)^2] \\ &= \text{E}[(\hat{\theta} - \text{E}[\hat{\theta}] + E]\hat{\theta}] - \theta)^2] \\ &= \text{E}[(\hat{\theta} - \text{E}[\hat{\theta}])^2] + 2\,\text{E}[(\hat{\theta} - \text{E}[\hat{\theta}])(\text{E}[\hat{\theta}] - \theta)] + \text{E}[(\text{E}[\hat{\theta}] - \theta)^2] \\ &= \text{E}[(\hat{\theta} - \text{E}[\hat{\theta}])^2] + \text{E}[(\text{E}[\hat{\theta}] - \theta)^2] \\ &= \text{Var}(\hat{\theta}) + \text{Bias}(\hat{\theta}) \end{aligned} \quad \text{(B.277)}$$

where the middle term is zero as $\text{E}[\hat{\theta} - \text{E}[\hat{\theta}]] = \text{E}[\hat{\theta}] - \text{E}[\hat{\theta}] = 0$.

The equation above indicates that the MSE is composed of two parts:

- *variance error* $\text{Var}(\hat{\theta}) = \text{E}[(\hat{\theta} - \text{E}[\hat{\theta}])^2]$:
  caused by the oversensitivity of a learning algorithm to small fluctuations in the training dataset due to random noise rather than the intended property of the data, a problem called *overfitting*.
- *bias error* $\text{Bias}(\hat{\theta}) = \text{E}[(\text{E}[\hat{\theta}] - \theta)^2]$:
  caused by the under-sensitivity of a learning algorithm to the dominating variation in the training dataset representing the essential relationship in the training dataset, a problem called *underfitting*.

In general, a proper bias-variance trade-off needs to be made for an algorithm to avoid either overfitting or underfitting, so that it can learn the essential relationship in the data while not affected by the inevitable noise in the data.

Notes

Chapter 2

1 Lewis Fry Richardson (1881–1953).

indexes