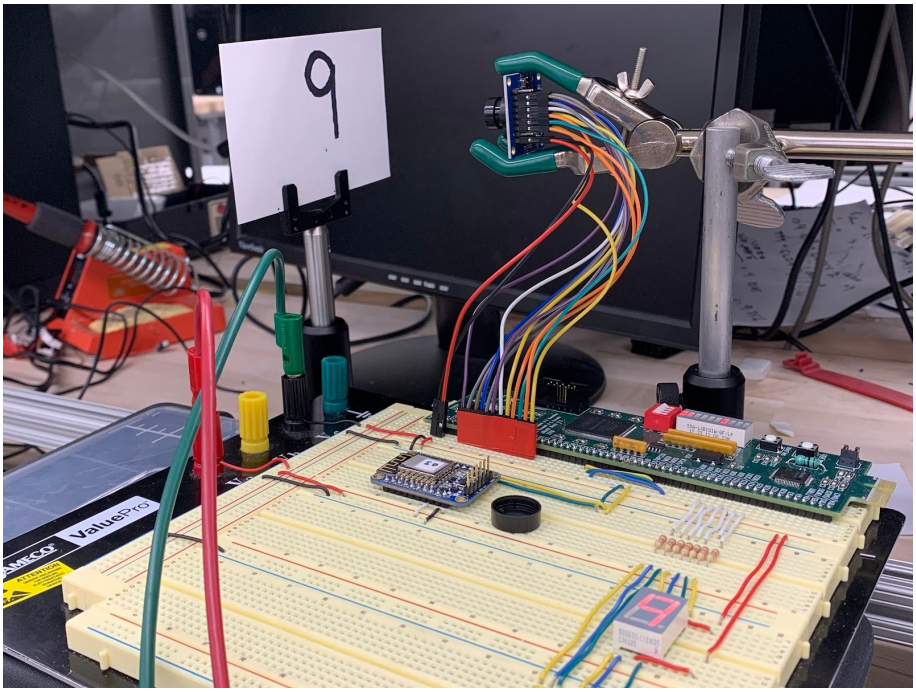
Neural Network Accelerator for Image Classification



Veronica Cortes and Richie Harris
E155: Microprocessor-Based Systems
David Harris and Josh Brake
13 December 2019

ABSTRACT

The goal of this project was to implement a neural network in hardware for image classification of handwritten digits. An image of a handwritten digit is captured by a camera and compressed on the FPGA. This compressed image enters a neural network which classifies the captured digit. The microcontroller receives this classification over SPI and displays the classified digit on a 7-segment display. The system correctly classified a set of digits 0 through 9 centered in the camera's field of vision. If the digit was centered in the camera's field of view and relatively thick, the system could correctly classify it almost every time.

TABLE OF CONTENTS

1. INTRODUCTION

The goal for our project was to build a system that can recognize and correctly classify handwritten digits 0 through 9 with at least 65% accuracy using a neural network accelerator on the FPGA.

1.1. MOTIVATION

This project was inspired by past coursework in Engineering Clinic and Machine Learning, where we both implemented neural networks to classify handwritten digits. In neither of these courses, however, were real handwritten digits used to test the neural network. This project gave us the opportunity to write digits ourselves, capture them using a camera, and visualize the classification in real-time.

There is also a rising interest in hardware accelerators for artificial intelligence applications such as robotics, IOT, and computer vision [4]. Hardware accelerators offer an advantage in speed when compared to software, and are thus useful for situations that necessitate processing large quantities of data quickly. This project served as an entry point into this exciting and emerging domain of digital hardware design.

1.2. OVERVIEW

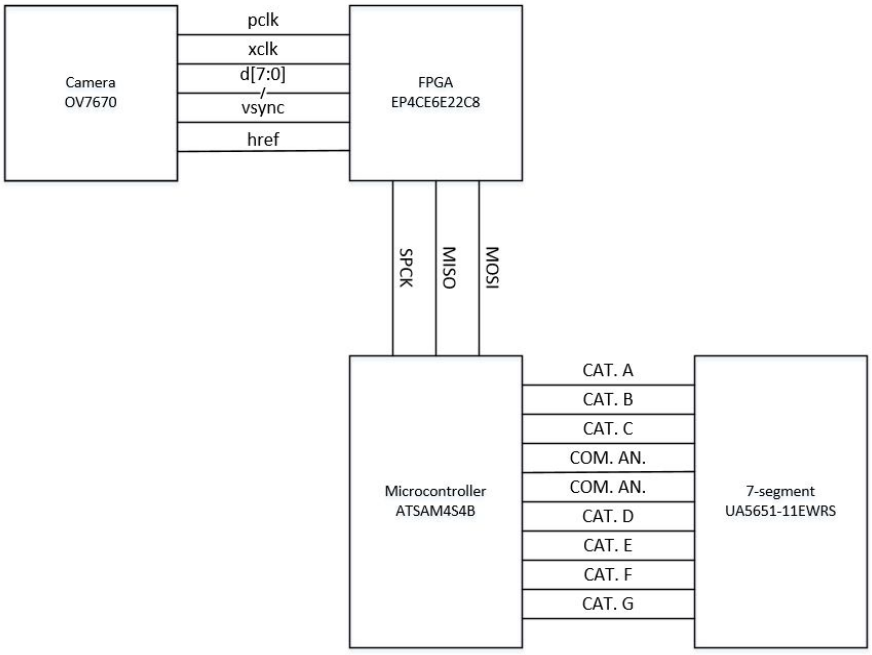


Figure 1. System-level block diagram

The overall block diagram of our system is shown in Figure 1. When the system is reset, the FPGA captures a new image from the camera through a parallel interface and decimates it to a smaller size so that it can be more easily processed by the neural network. The image is then classified by the neural network. After the neural network classifies the image, it sends the classification via SPI to the microcontroller. The microcontroller parses this information and illuminates a 7-segment display to show the digit that was classified. The breadboard schematic of the system is shown in Appendix B. The ESP8266 WIFI module included in the schematic was used for visualizing the camera images to debug the FPGA image capture module, but was not used in the final image classification system.

2. NEW HARDWARE

The new hardware for this project was the Omnivision OV7670 CMOS VGA camera module shown in Figure 2 below.

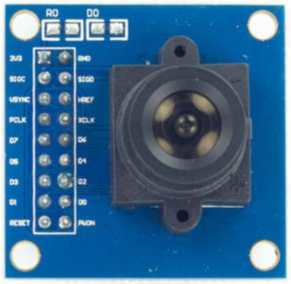


Figure 2. OV7670 camera module (Image: indiamart.com)

2.1. OV7670

The OV7670 is a low cost camera module capable of capturing VGA resolution images at up to 30 FPS. It contains an internal DSP chip that can pre-process the image to various resolutions, FPS rates, and image formats. These configurations can be set through the cameras SCCB interface [2]. For this project, we used the camera's default configuration of VGA (640x480) resolution, 30 FPS, and YCbCr image format. The pinout for the camera is shown in Table 1 below.

Pin	Type	Description
VDD**	Supply	Power supply
GND	Supply	Ground level
SDIOC	Input	SCCB clock
SDIOD	Input/Output	SCCB data
VSYNC	Output	Vertical synchronization
HREF	Output	Horizontal synchronization
PCLK	Output	Pixel clock
XCLK	Input	System clock
D0-D7	Output	Video parallel output
RESET	Input	Reset (Active low)
PWDN	Input	Power down (Active high)

Table 1. OV7670 pinout [3]

The camera operates on 3.3V power, 3.3V I/O, and does not contain an internal clock. Therefore, the user must supply the OV7670 a clock signal to the XCLK pin between 10 and 48 MHz. Then, the camera will drive its data pins D0-D7 and its synchronization pins VSYNC, HREF, and PCLK [3]. The timing diagrams for these signals for an entire image are shown in Figure 3 below.

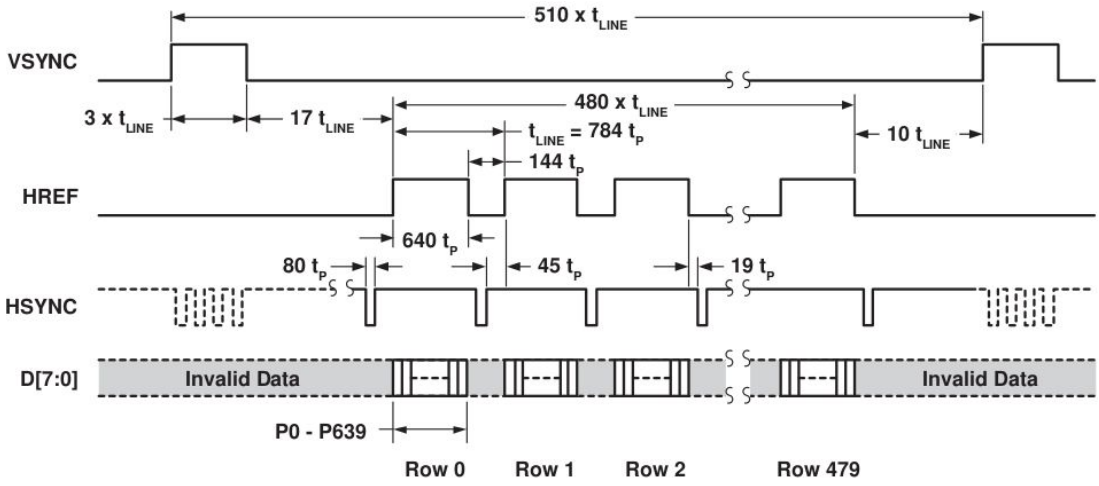


Figure 3. Timing diagram for an entire frame [2]

To indicate the start of a new frame, VSYNC goes high. Then, the image data is output row-wise starting from the top row. Each row of the image can be captured while HREF is high. Within a single row, the timing diagram is shown in Figure 4 below.

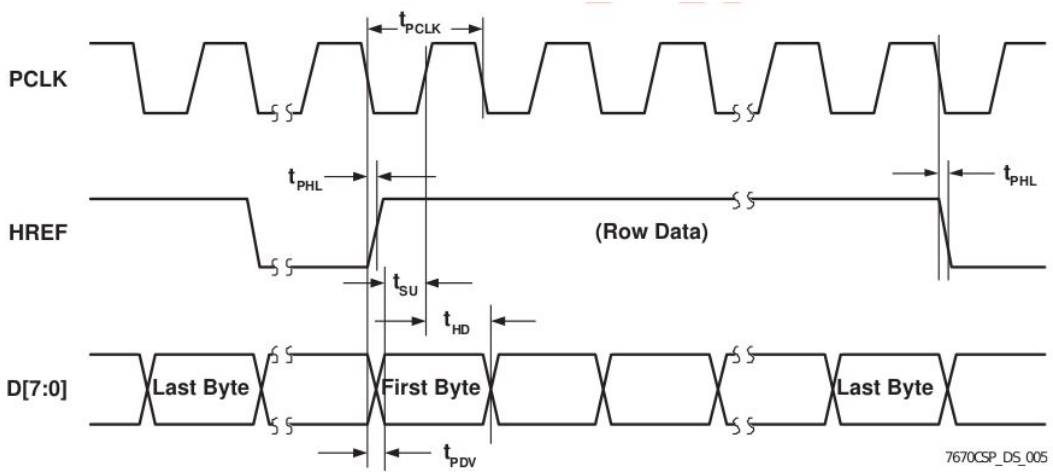


Figure 4. Timing diagram for a single row [2]

Each byte of data should be captured synchronously on the rising edge of PCLK. In each byte, D7 is the most significant bit and D0 is the least significant bit. In YCbCr format, Y is the luminance of the pixel, and Cb and Cr encode the color of the pixel. This data comes in the order shown in Table 2.

N	Byte
1st	Cb0
2nd	Y0
3rd	Cr0
4th	Y1
5th	Cb2
6th	Y2
7th	Cr2
8th	Y3
...	...

Pixel 0	Y0 Cb0 Cr0
Pixel 1	Y1 Cb0 Cr0
Pixel 2	Y2 Cb2 Cr2
Pixel 3	Y3 Cb2 Cr2
Pixel 4	Y4 Cb4 Cr4
Pixel 5	Y5 Cb4 Cr4

Table 2. Left: The order the bytes arrive in time. Right: How the pixels are constructed from these bytes. [3]

Each pixel is compressed such that Cb and Cr values are shared between every two adjacent pixels. For the purposes of image recognition of handwritten digits, we were only concerned with the luminance byte (Y) because we only needed the grayscale image. Therefore, we sampled every other byte from the parallel data pins.

3. MICROCONTROLLER SOFTWARE

The software on the microcontroller was designed to receive the classification from the FPGA over SPI, parse the classification, and display the classified digit on a 7-segment display.

3.1. DESIGN

The code implementing SPI communication was provided by the SAM4S4B library and the Lab 7 starter code. This code reads in the classification from the FPGA over SPI after the FPGA asserts 'done' by raising the `DONE_PIN` high.

The classified digit corresponds to the index of the maximum element in the classification array from the FPGA. The classification output by the FPGA is an array of 15 16-bit integers. The `spiSendReceive` method receives each byte of of this classification and stores it into an array of chars.

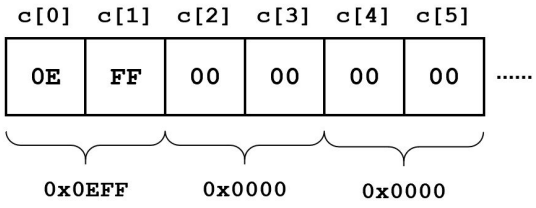


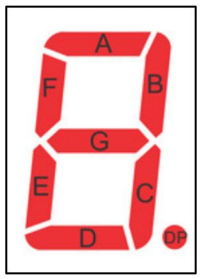
Figure 5. Classification array (in C) with 16-bit integer values identified

In order to find the maximum of this array, each 16-bit integer value must be calculated from adjacent char values, as shown in Figure 5. This is implemented using a for loop that iterates over the entire char array, finding each integer value according to Equation 1:

$$int16\ value = (classification[2i]\ <<\ 8)\ +\ classification[2i+1] \quad (1)$$

As the loop calculates each integer value, it keeps track of the maximum value seen so far and its index. When the loop breaks, it returns the index of the maximum value, which corresponds to the classified digit.

The classified digit is passed as an argument to `display_digit`, which finds the encoding of the segments for that digit (Figure 6) and displays the digit. These encodings are passed as strings for easier readability. C allows for easy conversion between chars and 8-bit integers, which allowed us to use the chars in the encoding string as integers in the write_segments method.



Hex	seg[6:0]
0	1000000
1	1111001
2	0100100
3	0110000
4	0011001
5	0010010
6	0000010
7	1111000
8	0000000
9	0011000
A	0001000
B	0000011
C	1000110
D	0100001
E	0000110
F	0001110

Figure 6. 7-segment encoding table where `seg[6:0]` corresponds to cathodes G through A (Image: mynewsdesk.com). A value of 0 pulls the corresponding cathode low, illuminating the segment.

3.2. TESTING

The microcontroller code was tested using the debugger and the 7-segment display itself. The debugger was used to validate that the classification array was parsed correctly, converting char values into corresponding 16-bit integer values. The 7-segment display was used to validate the segment encoding.

4. FPGA HARDWARE

The hardware on the FPGA is composed of two major subsystems: the image capture and the neural network. The camera hardware was designed to capture a frame from the camera and decimate it to a 16x16 image size. The neural network hardware implemented a 3-layer, 16-node feedforward neural network.

4.1. IMAGE CAPTURE

We designed the image capture module based on the OV7670 interface timing and encoding described in Section 2.1. The module is organized as a finite state machine with sequential logic that captures and decimates the image from VGA (640x480) to 16x16, storing the decimated image into a 1-D buffer. The general steps that the module performs are outlined as follows.

The process is started when VSYNC is asserted. Subsequently, on every rising PCLK edge where HREF is high, we can capture every other incoming byte of data corresponding to the luminance of the pixel. To compress the image to a size of 16x16, we maintain 16 accumulators. Since we have a VGA input (640x480) and the data arrives one row at a time, the first accumulator stores the sum of the first 40 (640/16) luminance bytes, the second accumulator stores the sum of the next 40 grayscale bytes, and so on until the end of the row. This procedure is repeated for a total of 30 (480/16) rows. The accumulated values are then right shifted by 10 to obtain values in the 8-bit range of 0-255, which correspond to the first row in the compressed 16x16 image. Repeating this for the rest of the image will yield the entire compressed 16x16 image. The decimation process is shown in Figure 7 below.

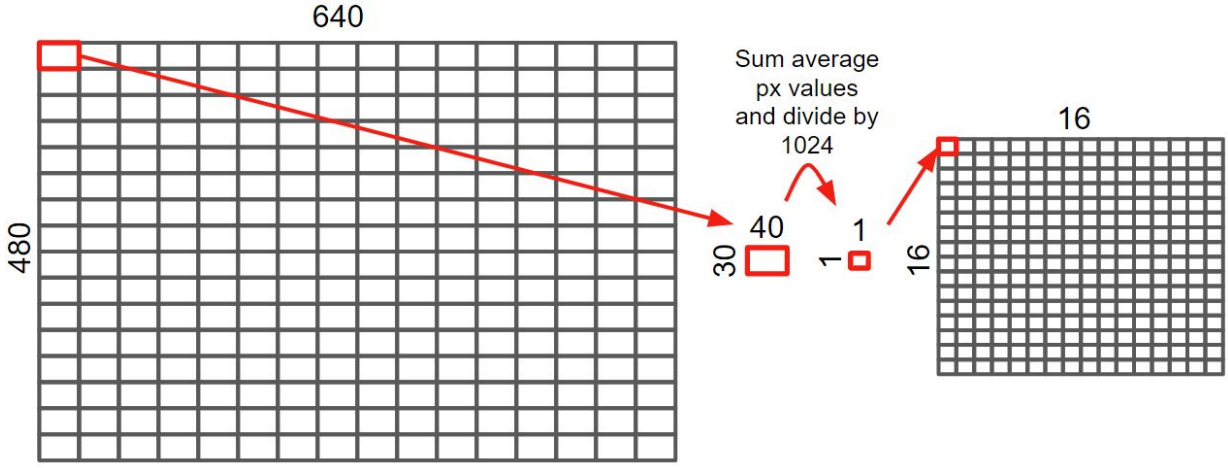


Figure 7. Decimation of 640x480 (VGA) to 16x16

4.2. NEURAL NETWORK

The neural network hardware implements a feedforward neural network composed of a 257-node input layer, two 16-node (15 nodes and 1 bias) hidden layers, and a 10-node output layer as shown in Figure 8.

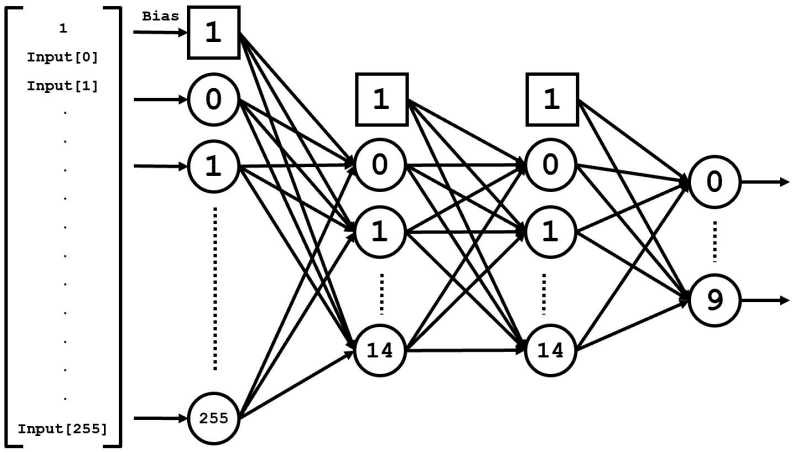


Figure 8. Neural network architecture

The input layer holds the 16x16 image data plus a bias of 1. Each hidden layer has 15 nodes (plus 1 bias) corresponding to the 15 multipliers available on the FPGA. The output layer has 10 nodes for the classification of the handwritten digit between 0 and 9.

4.2.1. DESIGN

The neural network architecture has been implemented with several memories (3 ROMs and 1 RAM) and two major subsystems: a datapath and a controller. The schematic for this architecture is in Appendix B.

Memories

The ROMs are used to store the weights for each hidden layer and the output layer. The stored weights are Q4.11 numbers. We could not train a neural network with accuracy greater than about 80% using the Q15 format. So, we chose the Q4.11 format as the best compromise between accuracy and resolution while avoiding overflow. Each row in ROM stores 15 Q4.11 numbers that feed the 15 execution units in the datapath; each cycle, each row of the weights is read out of ROM in sequence. The RAM is used to hold the output of the datapath on each cycle of execution.

The largest ROM is implemented as a synchronous ROM while the other memories used in the system are asynchronous. The FPGA will only synthesize synchronous ROMs and RAMs to memory blocks; asynchronous memories are synthesized using logic elements. The memories we implemented did not match the supported dimensions for the memory blocks on the FGPA. We could not get Quartus Prime to efficiency map our memories to the available memory blocks on the FPGA so implementing all synchronous memories exceeded the available memory blocks on the FGPA. At the same time, there are not enough logic elements to synthesize all of the memory blocks combinationally (including the rest of the neural network and the image capture subsystem). We chose to map the largest ROM to these memory blocks only to retain as many logic elements as possible while minimizing the complexity involved with having memory accesses occurring on different cycles for different memories.

Datapath

The datapath calculates the matrix multiplication for each layer over the course of many cycles as follows.

The datapath is composed of 15 execution units that multiply two Q4.11 numbers, accumulate their product, and activate the accumulated sum by applying the ReLU function to the sum. Given an 1xN input with Nx15 weights, each execution unit will calculate a column of the resulting 1x15 product over N+1 cycles, as shown in Figure 9.

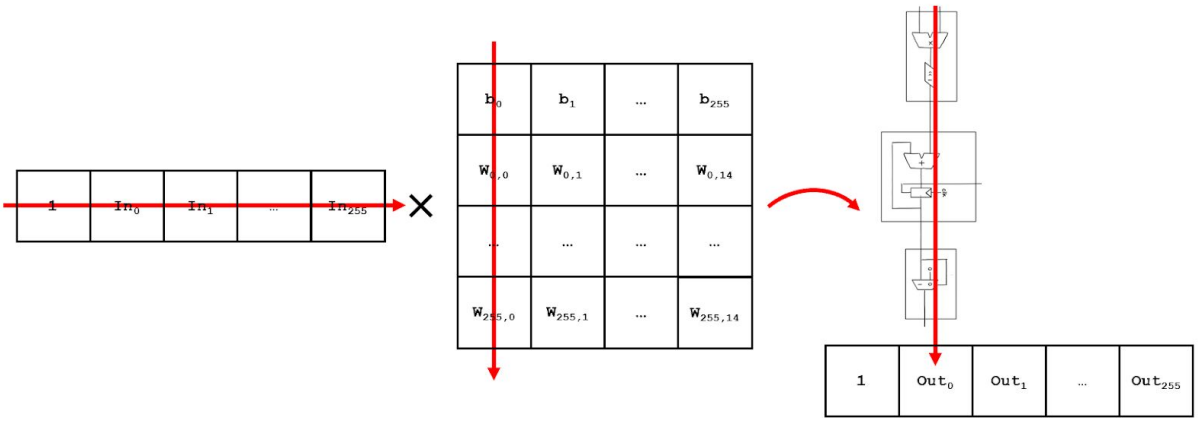


Figure 9. Matrix multiplication

After N+1 cycles, the product of each execution unit is concatenated into a bus of 15 Q4.11 numbers, corresponding to each of the columns of the product, that is stored in RAM for the computation of subsequent layers.

For our network architecture, the calculation proceeds as follows. First, the input image byte is converted to a Q4.11 value and scaled down to the range of 0-1 by padding with five zero bits in front of the byte and three zero bits at the end of the byte according to Equation 2.

```
assign px_int16 = {5'b0, px_uint8, 3'b0};
```
(2)

The first hidden layer is a 1x257 x 257x15 matrix multiplication and hence takes 258 cycles to accumulate the resulting product. Each hidden layer (and the output layer) is 1x16 x 16x15 matrix multiplication and hence takes 17 cycles to finish. Once the output layer has been calculated, the index of the maximum of the first 10 numbers of the output correspond to the classification.

Controller

The controller is a finite state machine which controls the source of the inputs to each execution unit (`src1` and `src2`), clears the accumulators at the end of each matrix multiplication, and sets the write enable for the RAM. Besides toggling these control signals, the controller also tracks which hidden layer is currently being operated on and, if the matrix multiplication is finished, delays execution for a cycle so that accumulators can be cleared.

The sources to each execution unit, `src1` and `src2`, are selected according to this table:

Layer	src1	src2
Hidden Layer 1	px_int16	rd1
Hidden Layer 2	prev_result	rd2
Output Layer	prev_result	rd3

Table 3. Execution source select

so that the appropriate input (the incoming pixel or the previous datapath output) and weights for each hidden layer match for the desired matrix multiplication. The RAM write enable is enabled on the last cycle of each matrix multiplication (and is otherwise disabled) to writeback the result in RAM. The accumulators are cleared on the following cycle before accumulating the products for the next multiplication.

4.2.2. TRAINING

We used MATLAB to train the neural network. Resources including code and training data were obtained from work done in a prior course in Machine Learning. The training data was a subset of the MNIST dataset consisting of 2240 16x16 images, with 224 images for each digit. The MNIST database contains 60,000 examples of 28x28 images of handwritten digits that is commonly used to train neural networks [1]. The pixel values ranged from 0 (black) to 255 (white). No modification was done to the training data because this range matched the 1 byte

values of the luminance from the camera. The code was modified to match our desired neural network architecture.

The network was trained by picking a random image from the training data, feeding it forward through the network, comparing the result to the expected classification, and backpropagating the error to update the weights. This process was repeated until the network achieved 99% accuracy when classifying all 2240 training images. After training the network, the weight vectors were converted to Q4.11 hexadecimal numbers and saved in .csv files by another MATLAB script. Once this was done, we used a Python script to convert these .csv files into .dat files with the correct format to be loaded into our SystemVerilog file.

4.3. TESTING

Both hardware subsystems were tested individually before they were integrated into the final system.

Image Capture
To test the image capture functionality, we sent the entire 16x16 decimated image over SPI to the microcontroller, which converted these bytes to integer values 0 - 255. Then, the ESP8266 WIFI module broadcasted a webpage containing these values so that they could easily be transferred to a text file on our computer. We created a MATLAB script to parse this text file and visualize the image. With this method, we were able to debug and test the image capture module to ensure that the captured images matched our expectations.

Neural Network
The neural network was tested in simulation using ModelSim Altera. The results of this simulation were compared against the output of the same neural network implemented in MATLAB. Using the MATLAB implementation, a set of expected values for each layer in the matrix was generated for a given test image. Using this same test image in the Verilog network, the output of the simulation at the end of each multiplication could be compared to the corresponding MATLAB output. We expect that the output of the network will deviate slightly from the MATLAB output due to rounding errors from our Q4.11 representation. Simulating the neural network in Modelsim, we saw that the output of the Verilog network was reasonably close to the MATLAB output. The waveforms for an example of this test are included in Appendix C.

5. RESULTS

The deliverables for the project were achieved. We implemented a system that could classify hand-written digits with high accuracy in real-time. While we did not perform formal testing to verify that the goal of 65% accuracy was met, we observed that if any well-written digit was positioned correctly in the image frame, the system could correctly classify it almost every time. Digits that represented the training data well were classified correctly with less than 1 error for every 10 classifications. Less well-written digits had an error rate closer to 1 in every 4 classifications. We decided not to execute a formal test for the accuracy because it would be very time-consuming to test the system on a sample size large enough to be representative of the different variations in handwriting.

The biggest sources of error in the classification were the digit being improperly centered in the frame or being too thin. Another potential source of error was differences between the training data and the experimental data captured from the camera. When testing, we noticed that the training data was much higher contrast than the data we obtained from the camera. Despite these challenges, our system could reliably identify all of the digits successfully under the right circumstances.

5.1. RECOMMENDATIONS FOR FUTURE WORK

While we were able to train a network with exceptionally high accuracy, the performance of the network on images outside of the training dataset was less accurate. One strategy to improve the accuracy of a handwritten digit was to draw it in the same style as the training data. Ideally, a user would be able to write a handwritten digit without any knowledge of the training data and get a correct classification. To this end, we could either 1) adapt the network to operate on 28x28 pixel images so that the entire MNIST dataset can be used for training or 2) compress images in the MNIST dataset to 16x16 pixels for use in the current network. Alternatively, a set of training data and labels could be developed by hand, but this would be very time consuming. To address the issue with the need to properly center the digit, we could incorporate training data with some spatial shifting.

In terms of modifying the design, the image capture subsystem can be improved. A processing step (renormalization, thresholding, etc.) could be added to prevent thinly written digits from being averaged out during decimation. Finally, adding a display to show the camera's frame of view would greatly ease the process of aiming the camera and improve user experience.

6. REFERENCES

[1] Y. LeCun, C. Cortes, and C. Burges, *The MNIST database of handwritten digits*, Accessed on: Dec. 10, 2019. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[2] *Omnivision Advanced Information preliminary datasheet,* Accessed on: Dec 10, 2019 [Online]. Available: http://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf

[3] J. Aparicio, *Hacking the OV7670 camera module (SCCB cheat sheet inside),* Accessed on: Dec 10, 2019 [Online]. Available: http://embeddedprogrammer.blogspot.com/2012/07/hacking-ov7670-camera-module-sccb-cheat.html

[4] J. Kobielus, *Powering AI: The Explosion of New AI Hardware Accelerators,* Accessed on: Dec 12, 2019 [Online]. Available: https://bereadycontenthub.com/beready/psg/art/powering-ai-the-explosion-of-new-ai-hardware-accelerators/

APPENDIX A

BILL OF MATERIALS

Part	Vender	Part No.	Price
OV7670 CMOS VGA camera module	Omnivision	OV7670	$10.99
ESP8266 WIFI module		ESP8266	
7-Segment Display		UA5651-11EWRS	

Parts listed without a vendor or a price were supplied to us by the Engineering Department Stockroom.

APPENDIX B

BREADBOARD SCHEMATIC

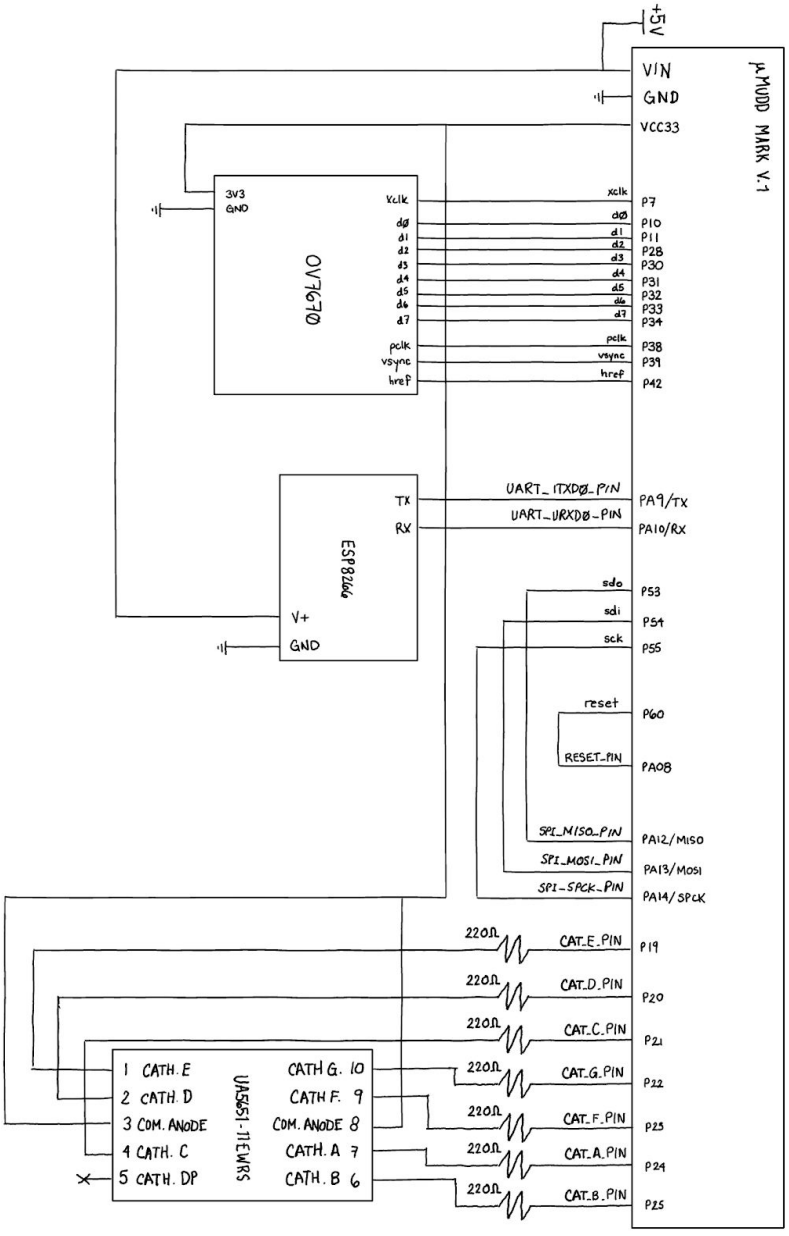


Figure B1. Breadboard schematic

NEURAL NETWORK SCHEMATIC

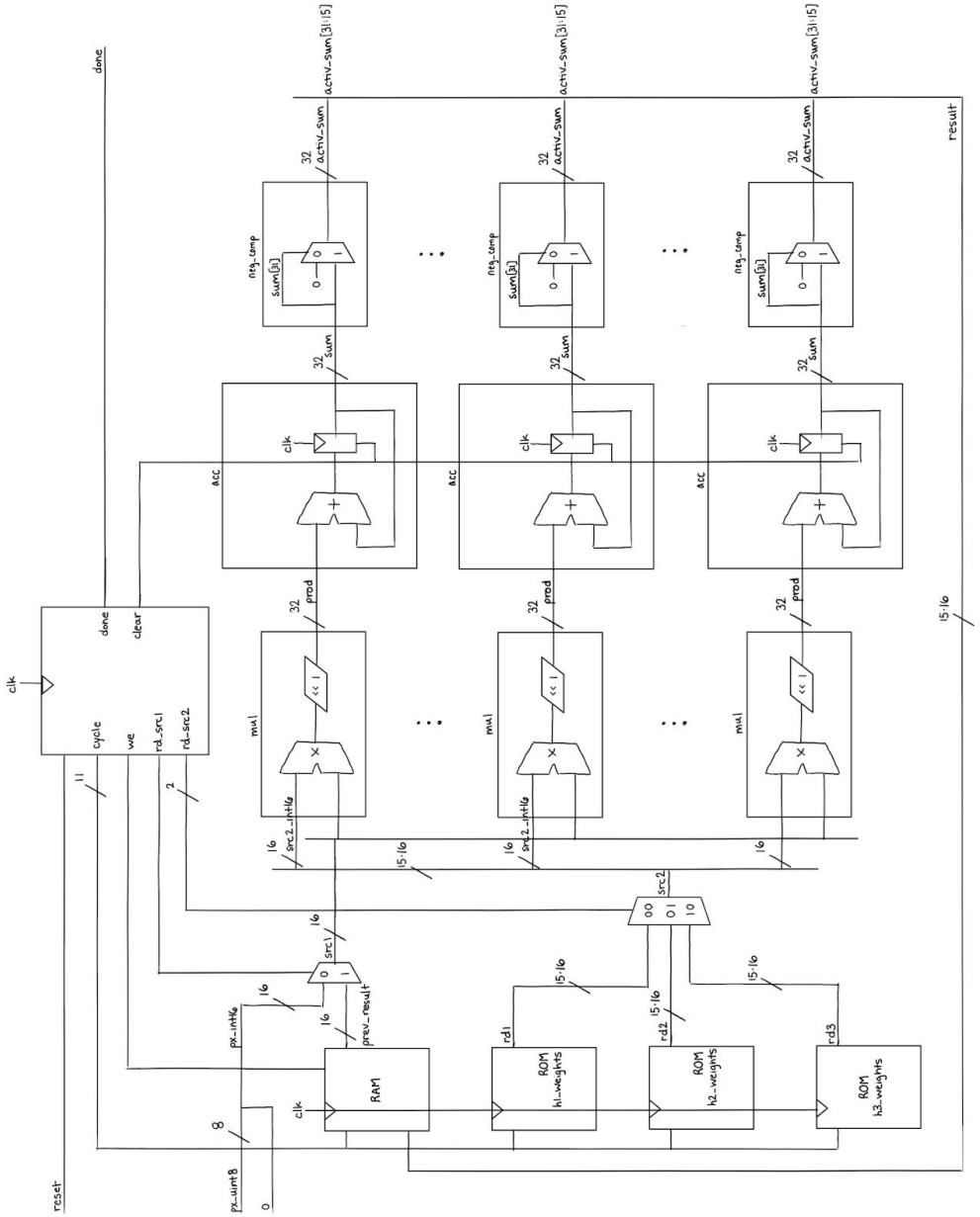
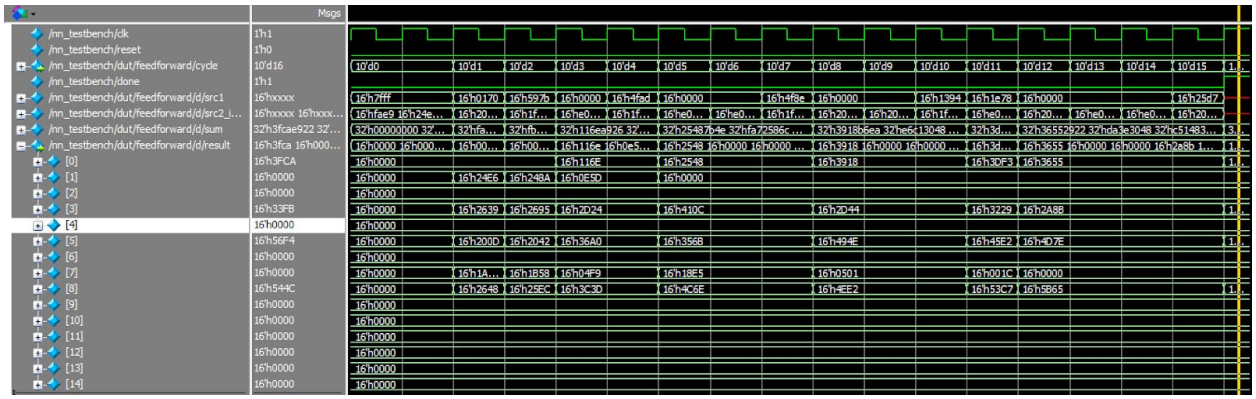


Figure B2. Neural network schematic

APPENDIX C

SIMULATION WAVEFORMS



APPENDIX D

The code appendices have been attached for greater readability. A public repository with the code for this project is maintained at the following URL:

`https://github.com/rkharris12/e155_final_project`

```
// rkharris@g.hmc.edu
// Richie Harris and Veronica Cortes
// receive a classification from the FPGA neural net


////////////////////////////////////////////////
// #includes
////////////////////////////////////////////////

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "SAM4S4B_lab7/SAM4S4B.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////

#define DONE_PIN  30
#define RESET_PIN   8
#define CAT_A_PIN PIO_PA24
#define CAT_B_PIN PIO_PA25
#define CAT_C_PIN PIO_PA21
#define CAT_D_PIN PIO_PA20
#define CAT_E_PIN PIO_PA19
#define CAT_F_PIN PIO_PA23
#define CAT_G_PIN PIO_PA22

////////////////////////////////////////////////
// Function Prototypes
////////////////////////////////////////////////

void reset_board(void);
void get_classification(char*);
char find_max_of_classification(char*);
void seven_segment_init(void);
void reset_segments(void);
void write_segments(char*);
void display_digit(char);

////////////////////////////////////////////////
// Main
////////////////////////////////////////////////

int main(void) {
  char classification[30]; // 2 bytes per node, 15 nodes

  samInit();
  pioInit();
  spiInit(MCK_FREQ/244000, 0, 1);
  // "clock divide" = master clock frequency / desired baud rate
  // the phase for the SPI clock is 1 and the polarity is 0
    tcInit();
    tcDelayInit();

  pioPinMode(DONE_PIN, PIO_INPUT);
    pioPinMode(RESET_PIN, PIO_OUTPUT);

    reset_segments(); // reset segments before init so they are initialized to 1
    seven_segment_init();

    reset_board();

  // recieve classification from FPGA
  get_classification(classification);
    char newDigit = find_max_of_classification(classification);
    display_digit(newDigit);

    while(1);
}

////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////

/* Toggles the reset pin used by the FGPA */
void reset_board(void) {
  pioDigitalWrite(RESET_PIN, 1);
  pioDigitalWrite(RESET_PIN, 0);
}

/* Writes classification received over SPI from FPGA to ATSAM local memory */
void get_classification(char *classification) {
  int i;

  while (!pioDigitalRead(DONE_PIN));

  for(i = 0; i < 30; i++) {
    classification[i] = spiSendReceive(0);
  }
}

/* Returns the classified digit */
char find_max_of_classification(char *classification) {
    int sum = 0;
    int new_max = 0;
    int index_of_new_max;
    for (int i = 0; i < 10; i++) {
        // Get MSB by shifting first element in classification array
        // LSB is second element in classification
        // Get 2B number by adding MSB and LSB
        sum = (classification[2*i] << 8) + classification[2*i+1];
```

```
        if (sum > new_max) {
            new_max = sum;
            index_of_new_max = i;
        }
    }
    return index_of_new_max + '0'; // convert to char
}

/* Set seven segment pins to PIO output mode */
void seven_segment_init(void) {
    pioPinMode(CAT_A_PIN, PIO_OUTPUT);
    pioPinMode(CAT_B_PIN, PIO_OUTPUT);
    pioPinMode(CAT_C_PIN, PIO_OUTPUT);
    pioPinMode(CAT_D_PIN, PIO_OUTPUT);
    pioPinMode(CAT_E_PIN, PIO_OUTPUT);
    pioPinMode(CAT_F_PIN, PIO_OUTPUT);
    pioPinMode(CAT_G_PIN, PIO_OUTPUT);
}

/* Initialize seven segment output to high (OFF) */
void reset_segments(void) {
    pioDigitalWrite(CAT_A_PIN, PIO_HIGH);
    pioDigitalWrite(CAT_B_PIN, PIO_HIGH);
    pioDigitalWrite(CAT_C_PIN, PIO_HIGH);
    pioDigitalWrite(CAT_D_PIN, PIO_HIGH);
    pioDigitalWrite(CAT_E_PIN, PIO_HIGH);
    pioDigitalWrite(CAT_F_PIN, PIO_HIGH);
    pioDigitalWrite(CAT_G_PIN, PIO_HIGH);
}

/* Write 7-segment cathodes using 7 digit string with segment encoding */
void write_segments(char * segments) {
    pioDigitalWrite(CAT_A_PIN, (int)(segments[6]-'0'));
    pioDigitalWrite(CAT_B_PIN, (int)(segments[5]-'0'));
    pioDigitalWrite(CAT_C_PIN, (int)(segments[4]-'0'));
    pioDigitalWrite(CAT_D_PIN, (int)(segments[3]-'0'));
    pioDigitalWrite(CAT_E_PIN, (int)(segments[2]-'0'));
    pioDigitalWrite(CAT_F_PIN, (int)(segments[1]-'0'));
    pioDigitalWrite(CAT_G_PIN, (int)(segments[0]-'0'));
}

/* Display the given digit on the 7-segment */
void display_digit(char digit) {

    // make array to hold segments
    char segments[8]; // 8 for 7 segments + null char

    // Look up 7-segment encoding for given digit
    switch(digit) {
        case '0':
            strcpy(segments, "1000000");
            break;
        case '1':
            strcpy(segments, "1111001");
            break;
        case '2':
            strcpy(segments, "0100100");
            break;
        case '3':
            strcpy(segments, "0110000");
            break;
        case '4':
            strcpy(segments, "0011001");
            break;
        case '5':
            strcpy(segments, "0010010");
            break;
        case '6':
            strcpy(segments, "0000010");
            break;
        case '7':
            strcpy(segments, "1111000");
            break;
        case '8':
            strcpy(segments, "0000000");
            break;
        case '9':
            strcpy(segments, "0011000");
            break;
        case 'A':
            strcpy(segments, "0001000");
            break;
        case 'B':
            strcpy(segments, "0000011");
            break;
        case 'C':
            strcpy(segments, "1000110");
            break;
        case 'D':
            strcpy(segments, "0100001");
            break;
        case 'E':
            strcpy(segments, "0000110");
            break;
        case 'F':
            strcpy(segments, "0001110");
            break;
        default:
            strcpy(segments, "1111111");
            break;
    }
    write_segments(segments);
```

```
}
```

```
/*
 * Authors: Veronica Cortes, Richie Harris
 * Email:   vcortes@g.hmc.edu, rkharris@g.hmc.edu
 * Date:    20 November 2019
 *
 * Feedforward neural network for image classification
 *
 */

`include "nn_15_node_defines.svh"

module top(input  logic     clk,
           input  logic     pclk, reset,
           input  logic     vsync,
           input  logic     href,
           input  logic     d0, d1, d2, d3, d4, d5, d6, d7,
           input  logic     sck,
           input  logic     sdi,
           output logic     sdo,
           output logic     done,
           output logic     xclk);

   assign xclk = clk; // drive camera xclk with 40 Hz from oscillator

   logic [2047:0]          frame;
   logic                   decimate_done;
   logic [`UINT_8-1:0]     px_uint8;
   logic [`ADR_LEN-1:0]    cycle, ra1;
   logic [15*16-1:0]       classification;

   decimate dec(pclk, reset, vsync, href, d0, d1, d2, d3, d4, d5, d6, d7, decimate_done, frame);

   spi s(sck, sdi, sdo, done, classification);

   choose_pixel cp(ra1, frame, px_uint8);

   nn feedforward(clk, reset, decimate_done, px_uint8, cycle, ra1, classification, done);

endmodule


module nn(input  logic                    clk, reset,
          input  logic                    decimate_done,
          input  logic [`UINT_8-1:0]      px_uint8,
          output logic [`ADR_LEN-1:0]     cycle, ra1,
          output logic [15*16-1:0]        classification,
          output logic                    done);

    // wires
    logic                                 we, clear;     // controls for RAM
    logic                                 rd_src1;
    logic [1:0]                           rd_src2;
    logic [`HIDDEN_LAYER_WID-1:0]         rd1, rd2, rd3; // rd from weight ROMs
    logic [0:`NUM_MULTS-1] [`INT_16-1:0]  result;        // wd to RAM
    logic [`RESULT_RD_WID-1:0]            prev_result;   // rd from RAM
```

```
56      logic                               captureclassification;
57
58      // weight memories
59      // 257 rows of 15 int16s
60      w1rom h1_weights(clk, ra1, rd1);
61      // 16 rows of 15 int16s
62      w2rom h2_weights(clk, cycle, rd2);
63      w3rom h3_weights(clk, cycle, rd3);
64
65      // output layer mem
66      oram result_ram(clk, we, cycle, result, prev_result);
67
68      // controller
69      controller c(clk, reset, decimate_done, we, cycle, ra1, rd_src1, rd_src2, clear, captureclassification);
70
71      // datapath
72      datapath d(clk, rd_src1, rd_src2, clear, px_uint8, rd1, rd2, rd3, prev_result, result, captureclassification,
    classification, done);
73
74
75  endmodule
76
77  module datapath(input  logic                                clk,
78                  input  logic                                rd_src1,
79                  input  logic [1:0]                          rd_src2,
80                  input  logic                                clear,
81                  input  logic [`UINT_8-1:0]                  px_uint8,
82                  input  logic [`HIDDEN_LAYER_WID-1:0]        rd1, rd2, rd3,
83                  input  logic [`RESULT_RD_WID-1:0]           prev_result,
84                  output logic [0:`NUM_MULTS-1] [`INT_16-1:0] result,
85                  input  logic                                captureclassification,
86                  output logic [15*16-1:0]                    classification,
87                  output logic                                done);
88
89      logic signed [`INT_16-1:0]                  px_int16;
90      logic signed [`INT_16-1:0]                  src1;
91      logic signed [`HIDDEN_LAYER_WID-1:0]        src2;
92      logic signed [0:`NUM_MULTS-1] [`INT_16-1:0] src2_int16;
93      logic signed [0:`NUM_MULTS-1] [`INT_32-1:0] prod, sum, activ_sum;
94      logic                                       captured;
95
96      // extend incoming image to int16
97      // maps [0,255] uint8 to [-16,16) Q4_11 int16
98      assign px_int16 = {5'b0, px_uint8, 3'b0};
99
100     // select read sources
101     /*  src1 | src2
102      *  -----------
103      *  img  | rd1
104      *  out  | rd2
105      *  out  | rd3
106      */
107      mux2 #(`INT_16) src1mux(px_int16, prev_result, rd_src1, src1);
108      mux3 #(`HIDDEN_LAYER_WID) src2mux(rd1, rd2, rd3, rd_src2, src2);
109
```

```
110       // generate datapath
111       genvar i;
112       generate
113           for (i=0 ; i<`NUM_MULTS; i++) begin : dpsl // generate wires
114            // 1st column of weights is the MSB of src2
115               // therefore, d0 corresponds to first hidden weight, d1 to second, ...
116               assign src2_int16[i] = src2[(`INT_16*(16-(i+1))-1) -: `INT_16]; //*(16-i) so that d0 is the first column (MSB)
117               mul #(`INT_16) m(src1, src2_int16[i], prod[i]);
118               acc #(`INT_32) lc(clk, clear, prod[i], sum[i]);
119               neg_comp #(`INT_32) relu(sum[i], activ_sum[i]);
120               assign result[i] = activ_sum[i][31:16];
121           end
122       endgenerate
123
124      // synchronously capture the classification when the output layer has been computed
125      always_ff @(posedge clk)
126         if (captureclassification & !captured) begin
127            classification <= {result[0], result[1], result[2], result[3], result[4], result[5], result[6], result[7], result
      [8], result[9], result[10], result[11], result[12], result[13], result[14]};
128            done <= 1'b1;
129            captured <= 1'b1;
130         end
131         else if (!captureclassification) begin
132            done <= 1'b0;
133            captured <= 1'b0;
134         end
135
136   endmodule
137
138   module controller(input  logic                clk, reset,
139                     input  logic                decimate_done,
140                     output logic                we,
141                     output logic [`ADR_LEN-1:0] cycle, ra1,
142                     output logic                rd_src1,
143                     output logic [1:0]          rd_src2,
144                     output logic                clear,
145                     output logic                captureclassification);
146
147      typedef enum logic [5:0] {RESET,
148                                MUL1, WB1, CLR1,
149                                MUL2, WB2, CLR2,
150                                MUL3, WB3, DONE} statetype;
151       statetype state, nextstate;
152
153       // control signals
154       logic [3:0] controls;
155       // flags
156       logic       input_layer_done, output_layer_done;
157       logic [1:0] layers_done_count;
158
159       always_ff @(posedge clk, posedge reset) begin
160          if (reset) state <= RESET;
161            else       state <= nextstate;
162       end
163
```

```
164       always_comb begin
165           case(state)
166               RESET: if (decimate_done)                 nextstate = MUL1;
167                      else                               nextstate = RESET;
168               MUL1:  if (cycle == `MULT_INPUT_CYCLES)    nextstate = WB1;
169                      else                               nextstate = MUL1;
170               WB1:                                      nextstate = CLR1;
171               CLR1:                                     nextstate = MUL2;
172               MUL2:  if (cycle == `MULT_HIDDEN_CYCLES)   nextstate = WB2;
173                      else                               nextstate = MUL2;
174               WB2:                                      nextstate = CLR2;
175               CLR2:                                     nextstate = MUL3;
176               MUL3:  if (cycle == `MULT_HIDDEN_CYCLES)   nextstate = DONE;
177                      else                               nextstate = MUL3;
178               DONE:                                     nextstate = DONE;
179               default:                                  nextstate = RESET;
180           endcase
181       end
182
183       // sequential controls
184       always_ff @(posedge clk)
185           case(state)
186               RESET:    begin
187                           cycle <= 0;
188                           layers_done_count <= 0;
189                         end
190               MUL1:        cycle <= cycle + 1'b1;
191               WB1:   begin
192                           cycle <= 0;
193                           layers_done_count <= layers_done_count + 1'b1;
194                         end
195               CLR1:        cycle <= 0; // delay to calculate final sum
196               MUL2:        cycle <= cycle + 1'b1;
197               WB2:   begin
198                           cycle <= 0;
199                           layers_done_count <= layers_done_count + 1'b1;
200                         end
201               CLR2:        cycle <= 0;
202               MUL3:        cycle <= cycle + 1'b1;
203               default:     cycle <= 0;
204           endcase
205
206      // combinational controls
207       always_comb begin
208         case(state)
209               RESET:   controls = 4'b0100;
210               MUL1:    controls = 4'b0000;
211               WB1:     controls = 4'b1010;
212               CLR1:    controls = 4'b0110;
213               MUL2:    controls = 4'b0010;
214               WB2:     controls = 4'b1010;
215               CLR2:    controls = 4'b0110;
216               MUL3:    controls = 4'b0010;
217               DONE:    controls = 4'b0011;
218               default: controls = 4'bxxxx;
```

```
219           endcase
220       end
221
222       // added for synchronous w1rom
223       always_comb begin
224           if (state == RESET & nextstate == MUL1) begin
225               ra1 = '0;
226           end
227           else if (state == MUL1) begin
228               ra1 = cycle + 1'b1;
229           end
230           else begin
231               ra1 = '0;
232           end
233       end
234
235
236        // set controls
237       assign {we, clear, input_layer_done, output_layer_done} = controls;
238
239        // assign flags
240       assign rd_src1 = input_layer_done;
241       assign rd_src2 = layers_done_count;
242       assign captureclassification    = output_layer_done;
243
244   endmodule
245
246   // selects a pixel from the frame buffer to input into the neural net
247   module choose_pixel(input  logic [`ADR_LEN-1:0] ra1,
248                       input  logic [2047:0]       frame,
249                       output logic [`UINT_8-1:0]  px_uint8);
250
251
252       logic                 select;
253       logic [`UINT_8-1:0]  bias;
254
255       assign bias = 8'hFF;
256
257       assign select = (ra1==10'd0 | ra1==10'd1);
258
259       mux2 #(`UINT_8) pixelmux(frame[`UINT_8*(ra1-1)-1 -: `UINT_8], bias, select, px_uint8);
260
261   endmodule
262
263
264   //
265   // --- NN Memories ---
266   //
267
268   // wh1rom is the only one implemented synchronously and thus actually
269   // stored to memory on the FPGA.
270   // The rest are stored in logic elements. The complexity was
271   // reduced when the roms were asynchronous so we chose to implement
272   // as few roms into memory as possible.
273   module w1rom(input  logic                          clk,
```

```
274                 input  logic [`ADR_LEN-1:0]          a,
275                 output logic [`HIDDEN_LAYER_WID-1:0] rd);
276
277       logic [`HIDDEN_LAYER_WID-1:0] ROM[`INPUT_LAYER_LEN-1:0];
278
279       initial
280           $readmemh("hiddenweights1.dat", ROM);
281
282       always_ff @(posedge clk)
283         rd <= ROM[a[`ADR_LEN-1:0]];
284   endmodule
285
286   module w2rom(input  logic                          clk,
287                input  logic [`ADR_LEN-1:0]          a,
288                output logic [`HIDDEN_LAYER_WID-1:0] rd);
289
290       logic [`HIDDEN_LAYER_WID-1:0] ROM[`HIDDEN_LAYER_LEN-1:0];
291
292       initial
293           $readmemh("hiddenweights2.dat", ROM);
294
295       assign rd = ROM[a[`ADR_LEN-1:0]];
296   endmodule
297
298
299   module w3rom(input  logic                          clk,
300                input  logic [`ADR_LEN-1:0]          a,
301                output logic [`HIDDEN_LAYER_WID-1:0] rd);
302
303       logic [`HIDDEN_LAYER_WID-1:0] ROM[`HIDDEN_LAYER_LEN-1:0];
304
305       initial
306           $readmemh("outputweights.dat", ROM);
307
308       assign rd = ROM[a[`ADR_LEN-1:0]];
309   endmodule
310
311   module oram(input  logic                        clk, we,
312               input  logic [`ADR_LEN-1:0]         a,
313               input  logic [`RESULT_WD_WID-1:0]   wd,
314               output logic [`RESULT_RD_WID-1:0]   rd);
315
316       logic [`RESULT_RD_WID-1:0] RAM[`RESULT_LEN-1:0];
317
318       assign rd = RAM[a[`ADR_LEN-1:0]];
319
320       always_ff @(posedge clk)
321           if (we) begin
322               RAM[0]  <= 16'h0800; // bias
323               RAM[1]  <= wd[`INT_16*15-1 -:`INT_16];
324               RAM[2]  <= wd[`INT_16*14-1 -:`INT_16];
325               RAM[3]  <= wd[`INT_16*13-1 -:`INT_16];
326               RAM[4]  <= wd[`INT_16*12-1 -:`INT_16];
327               RAM[5]  <= wd[`INT_16*11-1 -:`INT_16];
328               RAM[6]  <= wd[`INT_16*10-1 -:`INT_16];
```

```
329                 RAM[7]  <= wd[`INT_16*9-1  -:`INT_16];
330                 RAM[8]  <= wd[`INT_16*8-1  -:`INT_16];
331                 RAM[9]  <= wd[`INT_16*7-1  -:`INT_16];
332                 RAM[10] <= wd[`INT_16*6-1  -:`INT_16];
333                 RAM[11] <= wd[`INT_16*5-1  -:`INT_16];
334                 RAM[12] <= wd[`INT_16*4-1  -:`INT_16];
335                 RAM[13] <= wd[`INT_16*3-1  -:`INT_16];
336                 RAM[14] <= wd[`INT_16*2-1  -:`INT_16];
337                 RAM[15] <= wd[`INT_16*1-1  -:`INT_16];
338             end
339     endmodule
340
341     //
342     // --- NN-specific Gates ---
343     //
344
345     module mul #(parameter WIDTH = 16)
346                 (input  logic signed [WIDTH-1:0]   a, b,
347                  output logic signed [2*WIDTH-1:0] y);
348
349         assign y = (a * b) << 5; // LSL to get rid of extra integer bits
350
351     endmodule
352
353     module acc #(parameter WIDTH = 32)
354                 (input  logic                     clk, reset,
355                  input  logic signed [WIDTH-1:0] d,
356                  output logic signed [WIDTH-1:0] sum);
357
358         always_ff @(posedge clk, posedge reset)
359             if (reset) sum <= 0;
360             else       sum <= sum + d;
361     endmodule
362
363     module neg_comp #(parameter WIDTH = 8)
364                      (input  logic signed [WIDTH-1:0] x,
365                       output logic signed [WIDTH-1:0] y);
366
367         assign y = (x[WIDTH-1] == 0) ? x : '0;
368     endmodule
369
370     //
371     // --- Basic Logic Gates ---
372     //
373
374     module flopr #(parameter WIDTH = 8)
375                   (input  logic             clk, reset,
376                    input  logic [WIDTH-1:0] d,
377                    output logic [WIDTH-1:0] q);
378
379         always_ff @(posedge clk, posedge reset)
380             if (reset) q <= 0;
381             else       q <= d;
382     endmodule
383
```

```
384   module mux2 #(parameter WIDTH = 8)
385                (input  logic [WIDTH-1:0] d0, d1,
386                 input  logic             s,
387                 output logic [WIDTH-1:0] y);
388
389       assign y = s ? d1 : d0;
390   endmodule
391
392   module mux3 #(parameter WIDTH = 8)
393                (input  logic [WIDTH-1:0] d0, d1, d2,
394                 input  logic [1:0]       s,
395                 output logic [WIDTH-1:0] y);
396
397       assign y = s[1] ? d2 : (s[0] ? d1 : d0);
398   endmodule
399
400   //
401   // --- Camera modules ---
402   //
403
404   module decimate(input logic            pclk, reset,
405                   input logic            vsync,
406                   input logic            href,
407                   input logic            d0, d1, d2, d3, d4, d5, d6, d7,
408                   output logic           done,
409                   output logic [2047:0]  frame);
410
411      logic          y; // luminance is every other byte
412      logic [19:0]   a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15;
413      logic [9:0]    rowcount;
414      logic [9:0]    colcount;
415      logic [7:0]    min;
416      logic [11:0]   vsync_count;
417
418      typedef enum {RESET, COUNT, WAIT, START, GETROW, NEWROW, WAITROW, DECIMATE, LASTROW, DONE} statetype;
419      statetype state, nextstate;
420
421      always_ff @(posedge pclk, posedge reset) begin
422         if (reset)  state <= RESET;
423         else        state <= nextstate;
424      end
425
426      // next state logic
427      always_comb begin
428         case(state)
429                   // wait for vsync pulse at beginning of new frame
430            RESET:   if (vsync)                                          nextstate = COUNT;
431                     else                                                nextstate = RESET;
432                     // make sure vsync is indicating the start of a frame
433            COUNT:   if (~vsync)                                         nextstate = RESET;
434                     else if (vsync_count == `T_LINE_X3)                 nextstate = WAIT;
435                     else                                                nextstate = COUNT;
436            WAIT:    if (~vsync)                                         nextstate = START;
437                     else                                                nextstate = WAIT;
438                     // wait for href to start capture
```

```
439             START:   if (href)                                                  nextstate = GETROW;
440                      else                                                       nextstate = START;
441                      // capture a row until href goes low
442             GETROW:  if (~href)                                                 nextstate = NEWROW;
443                      else                                                       nextstate = GETROW;
444                      // end of row in decimated image
445                      // every 30 rows, shift a new decimated row into frame buffer
446                      // else wait for next row
447             NEWROW:  if (rowcount < `CAMERA_ROWS && (rowcount % `DEC_ROWS == 0)) nextstate = DECIMATE;
448                      else if (rowcount == `CAMERA_ROWS)                         nextstate = LASTROW;
449                      else                                                       nextstate = WAITROW;
450                      // wait for href
451             WAITROW: if (href)                                                  nextstate = GETROW;
452                      else                                                       nextstate = WAITROW;
453                      // shift out decimated bytes into frame buffer
454             DECIMATE:                                                          nextstate = WAITROW;
455                      // filter out bad images and shift out last decimated bytes
456             LASTROW: if (min > `MIN_THRESH)                                     nextstate = RESET;
457                      else                                                       nextstate = DONE;
458             DONE:                                                              nextstate = DONE;
459             default:                                                           nextstate = RESET;
460          endcase
461       end
462
463       // sequential image capture
464       always_ff @(posedge pclk) begin
465          if (state == RESET) begin
466             done <= 1'b0;
467             colcount <= '0;
468             y <= 1'b0;
469             min <= 8'b11111111;
470             vsync_count <= 12'b0;
471          end
472          // count how many clock cycles vsync is high to make sure it is the start of a new frame
473          else if (state == COUNT)   begin
474             vsync_count = vsync_count + 1'b1;
475          end
476          // reset colcount for new row
477          else if (state == WAIT) begin
478             done <= 1'b0;
479             colcount <= '0;
480             y <= 1'b0;
481          end
482          // reset rowcount, clear accumulators before start of new frame
483          else if (state == START) begin
484             rowcount <= '0;
485             {a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15} = '0;
486          end
487          // add chunks of 40 pixel values into each accumulator
488          else if (state == GETROW) begin
489             if (~y)  begin
490                if      (colcount <   `DEC_COLS)  a0  <= a0  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
491                else if (colcount < 2*`DEC_COLS)  a1  <= a1  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
492                else if (colcount < 3*`DEC_COLS)  a2  <= a2  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
493                else if (colcount < 4*`DEC_COLS)  a3  <= a3  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
```

```
494                else if (colcount < 5*`DEC_COLS)  a4  <= a4  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
495                else if (colcount < 6*`DEC_COLS)  a5  <= a5  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
496                else if (colcount < 7*`DEC_COLS)  a6  <= a6  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
497                else if (colcount < 8*`DEC_COLS)  a7  <= a7  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
498                else if (colcount < 9*`DEC_COLS)  a8  <= a8  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
499                else if (colcount < 10*`DEC_COLS) a9  <= a9  + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
500                else if (colcount < 11*`DEC_COLS) a10 <= a10 + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
501                else if (colcount < 12*`DEC_COLS) a11 <= a11 + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
502                else if (colcount < 13*`DEC_COLS) a12 <= a12 + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
503                else if (colcount < 14*`DEC_COLS) a13 <= a13 + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
504                else if (colcount < 15*`DEC_COLS) a14 <= a14 + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
505                else if (colcount < `CAMERA_COLS) a15 <= a15 + {12'b0, {d7, d6, d5, d4, d3, d2, d1, d0}};
506                colcount <= colcount + 1'b1;
507             end
508             y = y + 1'b1; // every other byte is the luminance
509             done <= 1'b0;
510          end
511          // finish row, increment rowcount, reset colcount
512          else if (state == NEWROW)  begin
513             rowcount <= rowcount + 1'b1;
514             colcount <= 10'b0;
515          end
516          // every time 30 rows are read, accumulators are full.  Shift out decimated bytes to frame buffer
517          else if (state == DECIMATE) begin
518             frame <= {{a15[17:10], a14[17:10], a13[17:10], a12[17:10], a11[17:10], a10[17:10], a9[17:10], a8[17:10], a7[17:10
    ], a6[17:10], a5[17:10], a4[17:10], a3[17:10], a2[17:10], a1[17:10], a0[17:10]}, frame[2047:128]};
519             {a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15} = '0;
520             if      (a0[17:10]  < min) min <= a0[17:10];
521             else if (a1[17:10]  < min) min <= a1[17:10];
522             else if (a2[17:10]  < min) min <= a2[17:10];
523             else if (a3[17:10]  < min) min <= a3[17:10];
524             else if (a4[17:10]  < min) min <= a4[17:10];
525             else if (a5[17:10]  < min) min <= a5[17:10];
526             else if (a6[17:10]  < min) min <= a6[17:10];
527             else if (a7[17:10]  < min) min <= a7[17:10];
528             else if (a8[17:10]  < min) min <= a8[17:10];
529             else if (a9[17:10]  < min) min <= a9[17:10];
530             else if (a10[17:10] < min) min <= a10[17:10];
531             else if (a11[17:10] < min) min <= a11[17:10];
532             else if (a12[17:10] < min) min <= a12[17:10];
533             else if (a13[17:10] < min) min <= a13[17:10];
534             else if (a14[17:10] < min) min <= a14[17:10];
535             else if (a15[17:10] < min) min <= a15[17:10];
536          end
537          // shift out last decimated row to the frame buffer
538          else if (state == LASTROW) begin
539             frame <= {{a15[17:10], a14[17:10], a13[17:10], a12[17:10], a11[17:10], a10[17:10], a9[17:10], a8[17:10], a7[17:10
    ], a6[17:10], a5[17:10], a4[17:10], a3[17:10], a2[17:10], a1[17:10], a0[17:10]}, frame[2047:128]};
540          end
541          // assert done
542          else if (state == DONE) begin
543              done <= 1'd1;
544          end
545       end
546
```

```
547   endmodule
548
549   ////////////////////////////////////////////
550   // spi
551   //   SPI interface.  Shifts out the classification
552   //   Tricky cases to properly change sdo on negedge clk
553   ////////////////////////////////////////////
554
555   module spi(input  logic sck,
556              input  logic sdi,
557              output logic sdo,
558              input  logic done,
559              input  logic [239:0] classification);
560
561       logic sdodelayed, wasdone;
562       logic [239:0] classificationcaptured;
563
564       // shift out the data.  80 scks to shift out data
565       always_ff @(posedge sck)
566           if (!wasdone)   classificationcaptured = classification;
567           else            classificationcaptured = {classificationcaptured[238:0], sdi};
568
569       // sdo should change on the negative edge of sck
570       always_ff @(negedge sck) begin
571           wasdone = done;
572           sdodelayed = classificationcaptured[238];
573       end
574
575       // when done is first asserted, shift out msb before clock edge
576       assign sdo = (done & !wasdone) ? classification[239] : sdodelayed;
577   endmodule
578
579
580
581
582
583
584
585
586
587
```

```
%% NeuralNet.m
%%
%% Richie Harris (rkharris@g.hmc.edu)
%%
%% Trains a 3-node, 16-layer neural network
%%


function main

    close all
    clear
    rng(4)

    X=load("data1.txt"); % 2240x256 dimensional, 10 class handwritten number data
    %csvwrite('images.csv', X);

    % labels for handwritten digit data, the location in the array
    % corresponds to the digit, first 224 are 0, next 224 are 1, ...
    Y=zeros(2240,10);
    Y(1:224,:)=[ones(224,1) zeros(224,9)];
    Y(225:448,:)=[zeros(224,1) ones(224,1) zeros(224,8)];
    Y(449:672,:)=[zeros(224,2) ones(224,1) zeros(224,7)];
    Y(673:896,:)=[zeros(224,3) ones(224,1) zeros(224,6)];
    Y(897:1120,:)=[zeros(224,4) ones(224,1) zeros(224,5)];
    Y(1121:1344,:)=[zeros(224,5) ones(224,1) zeros(224,4)];
    Y(1345:1568,:)=[zeros(224,6) ones(224,1) zeros(224,3)];
    Y(1569:1792,:)=[zeros(224,7) ones(224,1) zeros(224,2)];
    Y(1793:2016,:)=[zeros(224,8) ones(224,1) zeros(224,1)];
    Y(2017:2240,:)=[zeros(224,9) ones(224,1)];
    %csvwrite('labels.csv', Y);

    Yp=BackpropagationNetwork(X,Y); % (2240x10) get the results

    % convert Y and Yp to 1-D integer labels (1 to 10) so my confusion
    % matrix function can work on them
    Ynew=zeros(2240,1);
    Ypnew=zeros(2240,1);
    for i=1:2240
            Ynew(i)=find(Y(i,:)==1);
            Ypnew(i)=find(Yp(i,:)==max(Yp(i,:)));
    end
    [Cm error]=ConfusionMatrix(Ynew,Ypnew);
    Cm
    error

    % example image
    figure(1)
    showImage(X(500,:));
end



function Yp=BackpropagationNetwork(X,Y)
    [Nsamps D]=size(X); % Nsamps is number of samples (2240), D is dimension (256)
    N=D; M=10; % number of input layer nodes and output layer nodes, respectively
    L1=15; % number of hidden layer 1 nodes
    L2=15; % number of hidden layer 2 nodes
    % convert X to Q16.  Shift right by 16
    xscale=256;
    X = X/xscale;
    yscale = 1;
    Y=Y/yscale;
    wscale=100;
    Wh1new=(2*rand(N+1,L1)-ones(N+1,L1))/wscale; % (257xL1) randomly initialize N+1 dimensional (includes bias b) augmented
    hidden weight vectors Wh1=[wh1 wh2...whL].  Values between -0.25 and 0.25
    Wh2new=(2*rand(L1+1,L2)-ones(L1+1,L2))/wscale; % (L1+1xL2) randomly initialize L1+1 dimensional (includes bias b) augmented
    hidden weight vectors Wh2=[wh1 wh2...whL].  Values between -0.25 and 0.25
    Wonew=(2*rand(L2+1,M)-ones(L2+1,M))/wscale; % (L2+1x10) randomly initialize L2+1 dimensional (includes bias b) augmented
    output weight vectors Wo=[wo1 wo2...woM].  Values between -0.25 and 0.25
    eta=0.01; % learning rate
    tolerance=2*10^-2;
    error=inf;
    iter=0;
    dgh1 = ones(L1,1);
    dgh2 = ones(L2,1);
    dgo = ones(M,1);
    while error>tolerance
        iter=iter+1;
        Wh1old=Wh1new;
        Wh2old=Wh2new;
        Woold=Wonew;
        n=randi(Nsamps);
        xtrain=X(n,:)'; % (256x1) randomly select a training sample
        xtrain=[1;xtrain]; % (257x1) D+1 augmented training sample to include bias b=1
        ytrain=Y(n,:); % (1x10) randomly selected training sample's corresponding label
%         flag=0; % overflow flag

        % forward pass
        ah1=Wh1old'*xtrain; % activation (net input) of hidden layer 1
        for i=1:length(ah1) % Relu activation function
%             if (ah1(i) >= 1) || (ah1(i) <= -1) % prevent overflow
%                 flag=1;
%                 break;
%             end
            if ah1(i) < 0
                ah1(i) = 0;
                dgh1(i) = 0;
            else
                dgh1(i) = 1;
            end
        end
```

```
%         if flag==1
%             break;
%         end
          z1=[1;ah1]; % augmented output of hidden layer 1
          ah2=Wh2old'*z1; % activation (net input) of hidden layer 2
          for i=1:length(ah2) % Relu activation function
%               if (ah2(i) >= 1) || (ah2(i) <= -1) % prevent overflow
%                   flag=1;
%                   break;
%               end
              if ah2(i) < 0
                  ah2(i) = 0;
                  dgh2(i) = 0;
              else
                  dgh2(i) = 1;
              end
          end
%         if flag==1
%             break;
%         end
          z2=[1;ah2]; % augmented output of hidden layer 2
          ao=Woold'*z2; % activation (net input) of output layer
          for i=1:length(ao) % Relu activation function
%               if (ao(i) >= 1) || (ao(i) <= -1) % prevent overflow
%                   flag=1;
%                   break;
%               end
              if ao(i) < 0
                  ao(i) = 0;
                  dgo(i) = 0;
              else
                  dgo(i) = 1;
              end
          end
%         if flag==1
%             break;
%         end
          yp=ao'; % output of output layer

          % backward error propagation
          % 2 layers
          do=(ytrain'-yp').*dgo; % find d of output layer (Mx1 vector)
          dh2=(Woold(2:L2+1,1:M)*do).*dgh2; % find d of hidden layer 2 (L2x1 vector).  Remove the first row of Wo: the bias offset
          dh1=(Wh2old(2:L1+1,1:L2)*dh2).*dgh1; % find d of hidden layer 1 (L1x1 vector).  Remove the first row of Wh2: the bias
          offset

          Wonew=Woold+(eta*do*z2')'; % update weights of output layer
          Wh2new=Wh2old+(eta*dh2*z1')'; % update weights of hidden layer 2
          Wh1new=Wh1old+(eta*dh1*xtrain')'; % update weights of hidden layer 1

%         if ~any(do) || ~any(dh3) || ~any(dh2) || ~any(dh1) % check if the network is being trained properly
%             break;
%         end

%         if min(min(Wh1new))<-1 || max(max(Wh1new))>1 || min(min(Wh2new))<-1 || max(max(Wh2new))>1 || min(min(Wonew))<-1 ||
max(max(Wonew))>1 % check for overflow
%             break;
%         end

          if ~mod(iter,100000) % check error every 100000 iterations
              % 2 layer
              hidden1 = relu(Wh1new'*[ones(Nsamps,1) X]');
              hidden2 = relu(Wh2new'*[ones(1,Nsamps); hidden1]);
              Yp=relu(Wonew'*[ones(1,Nsamps); hidden2])'; % forward pass to get output Yp given X
              % convert Y and Yp to 1-D integer labels so my confusion matrix
              % function can work on them
              Ynew=zeros(Nsamps,1);
              Ypnew=zeros(Nsamps,1);
              for i=1:Nsamps
                  Ynew(i)=find(Y(i,:)==1/yscale);
                  val = find(Yp(i,:)==max(Yp(i,:)));
                  if length(val) ~= 1
                      Ypnew(i) = randi(10);
                  else
                      Ypnew(i) = val;
                  end
              end
              [Cm error]=ConfusionMatrix(Ynew,Ypnew);
          end
      end
      % 2 layer
      hidden1 = relu(Wh1new'*[ones(Nsamps,1) X]');
      hidden2 = relu(Wh2new'*[ones(1,Nsamps); hidden1]);
      Yp=relu(Wonew'*[ones(1,Nsamps); hidden2])'; % forward pass to get output Yp given X
  end


  function ret=relu(nodes) % compute relu for nodes for all samples (nodes is 30x2240 for hidden)
      [D Nsamps]=size(nodes); % Nsamps is number of samples, D is dimension
      ret = ones(D, Nsamps);
      for i = 1:D
         for j = 1:Nsamps
             if nodes(i,j) < 0
                 ret(i,j) = 0;
             else
                 ret(i,j) = nodes(i,j);
             end
         end
      end
  end
```

```
function showImage(mk) % displays image.  Input is a single row of X (1x256)
    % convert x to a value from 0 to 1
    xmin=min(mk);
    xmax=max(mk);
    s=1/(xmax-xmin);
    for i=1:(length(mk))
        mk(i)=(mk(i)-xmin)*s;  % Convert to entire dynamic range
    end
    m=reshape(mk,[16 16]);
    hold on
    for i=1:16
        for j=1:16
            val=m(i,j);
            scatter(i,17-j,1000,[val val val],'filled','s');
        end
    end
    hold off
end



% calculate confusion matrix and error rate
function [Cm er]=ConfusionMatrix(y,yp)
    N=length(y);             % number of samples
    K=length(unique(y));     % number of classes
    Cm=zeros(K);             % initialize confusion matrix
    for n=1:N
        Cm(y(n),yp(n))=Cm(y(n),yp(n))+1; % fill in confusion matrix
    end
    er=1-trace(Cm)/sum(sum(Cm)); % er=0 means 0% error.  All classifications are correct
end

```

```
%% classify.m
%%
%% Richie Harris (rkharris@g.hmc.edu)
%% Veronica Cortes (vcortes@g.hmc.edu)
%%
%% Computes output layer of 3-layer, 16-node neural network using weights from NeuralNet.make
%% Converts weights and layers into Q4.11 values and writes to .csv files
%%

wh1 = Wh1old;
wh2 = Wh2old;
wh3 = Wh3old;
wo = [Woold zeros(16,5)];
Wh1new = Wh1old;
Wh2new = Wh2old;
Wh3new = Wh3old;
Wonew = Woold;
% make cells
wh1c = num2cell(arrayfun(@dec2q,wh1));
wh2c = num2cell(arrayfun(@dec2q,wh2));
wh3c = num2cell(arrayfun(@dec2q,wh3));
woc = num2cell(arrayfun(@dec2q,wo));
% write csv's
csvwrite('wh1.csv', wh1);
csvwrite('wh2.csv', wh2);
csvwrite('wh3.csv', wh3);
csvwrite('wo.csv', wo);
cell2csv('wh1_q15.csv', wh1c);
cell2csv('wh2_q15.csv', wh2c);
cell2csv('wh3_q15.csv', wh3c);
cell2csv('wo_q15.csv', woc);

X=load("data1.txt");
xtrain = X(2,:)';
xtrain = [255;xtrain]; % do 255 because that will become effectively 1 when we do dec2hex
xconverted = dec2hex(xtrain);
xt = cellstr(xconverted);
csvwrite('xtrain.csv', xtrain);
cell2csv('xtrain_q15.csv', xt);

xtrain = xtrain/1024; % did /512 in TF
ah1=Wh1new'*xtrain; % activation (net input) of hidden layer 1
h1 = num2cell(arrayfun(@dec2q,ah1));
csvwrite('h1.csv', ah1);
cell2csv('h1_q15.csv', h1);
for i=1:length(ah1) % Relu activation function
    if ah1(i) < 0
        ah1(i) = 0;
    end
end
reluh1 = num2cell(arrayfun(@dec2q,ah1));
z1=[1;ah1]; % augmented output of hidden layer 1
ah2=Wh2new'*z1; % activation (net input) of hidden layer 2
h2 = num2cell(arrayfun(@dec2q,ah2));
csvwrite('h2.csv', ah2);
cell2csv('h2_q15.csv', h2);
for i=1:length(ah2) % Relu activation function
    if ah2(i) < 0
        ah2(i) = 0;
    end
end
reluh2 = num2cell(arrayfun(@dec2q,ah2));
z2=[1;ah2]; % augmented output of hidden layer 1
ah3=Wh3new'*z2; % activation (net input) of hidden layer 2
h3 = num2cell(arrayfun(@dec2q,ah3));
csvwrite('h3.csv', ah3);
cell2csv('h3_q15.csv', h3);
for i=1:length(ah3) % Relu activation function
    if ah3(i) < 0
        ah3(i) = 0;
    end
end
z3=[1;ah3]; % augmented output of hidden layer 2
ao=Wonew'*z3; % activation (net input) of output layer
ol = num2cell(arrayfun(@dec2q,ao));
csvwrite('ol.csv', ao);
cell2csv('ol_q15.csv', ol);
for i=1:length(ao) % Relu activation function
    if ao(i) < 0
        ao(i) = 0;
    end
end
yp=ao'; % output of output layer

expected = num2cell(arrayfun(@dec2q,yp));
csvwrite('expected.csv', yp);
cell2csv('expected_q15.csv', expected);

%% Do this for 2 layer 15 node networks
clear
load weights_2layers_15nodes_99percent.mat % this overflows -1 to 1.  But minimum value is -6.  Max is 4.  So use Q3_12 => 4
integer, 12 decimal

wh1 = Wh1new;
wh2 = Wh2new;
wo = [Wonew zeros(16,5)];
% make cells
wh1c = num2cell(arrayfun(@(o) dec2q(o,4,11), wh1));
wh2c = num2cell(arrayfun(@(o) dec2q(o,4,11), wh2));
woc = num2cell(arrayfun(@(o) dec2q(o,4,11), wo));
% write csv's
csvwrite('wh1.csv', wh1);
```

```
csvwrite('wh2.csv', wh2);
csvwrite('wo.csv', wo);
cell2csv('wh1_q.csv', wh1c);
cell2csv('wh2_q.csv', wh2c);
cell2csv('wo_q.csv', woc);

X=load("data1.txt");
xtrain = X(2,:)';
xtrain = [255;xtrain]; % do 255 because that will become effectively 1 when we do dec2hex
xconverted = dec2hex(xtrain);
xt = cellstr(xconverted);
csvwrite('xtrain.csv', xtrain);
cell2csv('xtrain_hex.csv', xt);

xtrain = xtrain/256;
ah1=Wh1new'*xtrain; % activation (net input) of hidden layer 1
h1 = num2cell(arrayfun(@(o) dec2q(o,4,11), ah1));
csvwrite('h1.csv', ah1);
cell2csv('h1_q.csv', h1);
for i=1:length(ah1) % Relu activation function
    if ah1(i) < 0
        ah1(i) = 0;
    end
end
reluh1 = num2cell(arrayfun(@(o) dec2q(o,4,11), ah1));
z1=[1;ah1]; % augmented output of hidden layer 1
ah2=Wh2new'*z1; % activation (net input) of hidden layer 2
h2 = num2cell(arrayfun(@(o) dec2q(o,4,11), ah2));
csvwrite('h2.csv', ah2);
cell2csv('h2_q.csv', h2);
for i=1:length(ah2) % Relu activation function
    if ah2(i) < 0
        ah2(i) = 0;
    end
end
reluh2 = num2cell(arrayfun(@(o) dec2q(o,4,11), ah2));
z2=[1;ah2]; % augmented output of hidden layer 1
ao=Wonew'*z2; % activation (net input) of output layer
ol = num2cell(arrayfun(@(o) dec2q(o,4,11), ao));
csvwrite('ol.csv', ao);
cell2csv('ol_q.csv', ol);
for i=1:length(ao) % Relu activation function
    if ao(i) < 0
        ao(i) = 0;
    end
end
yp=ao'; % output of output layer

expected = num2cell(arrayfun(@(o) dec2q(o,4,11), yp));
csvwrite('expected.csv', yp);
cell2csv('expected_q.csv', expected);

```

```python
# Veronica Cortes
# vcortes@g.hmc.edu
# 16 November 2019

# ----------------
# Binary Math
# ----------------

import math

def num2bin(num, str):
    if (int(num) == 0):
        while(len(str) < 8):
            str = "0" + str
        return str
    elif (int(num) == 1):
        str = "1" + str
    elif (int(num) % 2 == 1):
        str = "1" + str
    else:
        str = "0" + str
    return num2bin(int(num)/2, str)

def bin2hex(numStr):
    hx = ''

    conv = {
        '0000': '0',
        '0001': '1',
        '0010': '2',
        '0011': '3',
        '0100': '4',
        '0101': '5',
        '0110': '6',
        '0111': '7',
        '1000': '8',
        '1001': '9',
        '1010': 'A',
        '1011': 'B',
        '1100': 'C',
        '1101': 'D',
        '1110': 'E',
        '1111': 'F'
    }

    parts = [numStr[i:i+4] for i in range(0,len(numStr),4)]
    for e in parts:
        hx += conv[e]
    return hx

def xorStr(strA, strB):
    newStr = ''
    tt = {
        '00': '0',
        '01': '1',
        '10': '1',
        '11': '0',
    }
    AB = ''
    for i in range(0,len(strA)):
        AB = strA[i] + strB[i]
        newStr += tt[AB]
    return newStr

def addBinOne(binStr):
    revBinStr = binStr[::-1]
    c = 0
    newStr = ''
    if (revBinStr[0] == '1'):
        newStr = '0' + newStr
        c = 1
    elif (revBinStr[0] == '0'):
        newStr = '1' + newStr
    for i in range(1,len(revBinStr)):
        if (revBinStr[i] == '1' and c == 1):
            newStr = '0' + newStr
        elif (revBinStr[i] == '0' and c == 1):
            newStr = '1' + newStr
            c = 0
        elif (revBinStr[i] == '1' and c == 0):
            newStr = '1' + newStr
        else:
            newStr = '0' + newStr
    return newStr

def twosComp(binStr):
    #return (num^-1)+1;
    return addBinOne(xorStr(binStr,'1111111111111111'))

def hex2(num):
    if (num < 16):
        return '0x0' + hex(num)[2::]
    else:
        return hex(num)

def bin2frac(binStr):
    str = binStr
    exp = -1
    sum = 0
    if (binStr[0] == '1'):
        str = twosComp(str)
        sum += -1
```

```
    for e in binStr[1::]:
        sum += float(e)*pow(2, exp)
        exp -= 1
    return sum

def frac2bin(num):
    str = ''
    mag = abs(num)
    exp = -1
    count = 0
    while (mag > 0 and count < 7):
        if mag >= pow(2,exp):
            str += '1'
            mag -= pow(2,exp)
        else:
            str += '0'
        #print(mag)
        exp -= 1
        count += 1
    str = '0' + str
    if (num < 0):
        str = twosComp(str)
    while (len(str) < 8):
        str += '0'
    #print(str)
    return str

# ----------------
# Load in CSVs
# ----------------

import csv

inputLayer = []
hiddenWeights1 = []
hiddenWeights2 = []
hiddenWeights3 = []
outputWeights = []
newrow = []

with open('xtrain_q15.csv', newline='') as inputLayerFile:
    inputLayerRead = csv.reader(inputLayerFile, delimiter=' ', quotechar='|')
    for row in inputLayerRead:
        row = row[0].split(',')
        for e in row:
            newrow += [e]
        inputLayer += [newrow]
        newrow = []

with open('wh1_q15.csv', newline='') as hiddenWeights1File:
    hiddenWeights1Read = csv.reader(hiddenWeights1File, delimiter=' ', quotechar='|')
    for row in hiddenWeights1Read:
        row = row[0].split(',')
        for e in row:
            newrow += [e]
        hiddenWeights1 += [newrow]
        newrow = []

with open('wh2_q15.csv', newline='') as hiddenWeights2File:
    hiddenWeights2Read = csv.reader(hiddenWeights2File, delimiter=' ', quotechar='|')
    for row in hiddenWeights2Read:
        row = row[0].split(',')
        for e in row:
            newrow += [e]
        hiddenWeights2 += [newrow]
        newrow = []

with open('wh3_q15.csv', newline='') as hiddenWeights3File:
    hiddenWeights3Read = csv.reader(hiddenWeights3File, delimiter=' ', quotechar='|')
    for row in hiddenWeights3Read:
        row = row[0].split(',')
        for e in row:
            newrow += [e]
        hiddenWeights3 += [newrow]
        newrow = []

with open('wo_q15.csv', newline='') as outputWeightsFile:
    outputWeightsRead = csv.reader(outputWeightsFile, delimiter=' ', quotechar='|')
    for row in outputWeightsRead:
        row = row[0].split(',')
        for e in row:
            newrow += [e]
        outputWeights += [newrow]
        newrow = []

# ----------------
# Write DATs
# ----------------

f = open("inputlayer.dat","w+")

out = ''
for r in range(0, len(inputLayer)):
    f.write(out.join(inputLayer[r]) + "\r\n");
    out = ''

f.close()

f = open("hiddenweights1.dat","w+")

out = ''
for r in range(0, len(hiddenWeights1)):
```

```
        f.write(out.join(hiddenWeights1[r]) + "\r\n");
        out = ''

    f.close()

    f = open("hiddenweights2.dat","w+")

    out = ''
    for r in range(0, len(hiddenWeights2)):
        f.write(out.join(hiddenWeights2[r]) + "\r\n");
        out = ''

    f.close()

    f = open("hiddenweights3.dat","w+")

    out = ''
    for r in range(0, len(hiddenWeights3)):
        f.write(out.join(hiddenWeights3[r]) + "\r\n");
        out = ''

    f.close()

    f = open("outputweights.dat","w+")

    out = ''
    for r in range(0, len(outputWeights)):
        f.write(out.join(outputWeights[r]) + "\r\n");
        out = ''

    f.close()

```