Final Report

Timothy Baalman

ECEN 4303

**Introduction:**

This was a very interesting project but due to the complexity of the project I did not get as much done as I initially intended to. I have the Network fully built and tested verifying that the correct digit was detected for the loaded in picture values. So, I did complete the project, but I did not get around to implementing the dadda-multiplier, the CLA, or extra layers. I do have a tested CLA module though. I also implemented a softmax with before and after waveforms of the network. There was some optimization complexity of connecting everything together that I will discuss later, but that required optimization took a long amount of time implementing. Which is why I did not get to the dadda-multiplier, or the CLA. I made my own custom float to fixed-point converter as well as a fixed-point to float converter. The later one was needed to verify results of the 32-bit adder and 32-bit multiplier. I wrote my code in such a way that increasing the bit address to 64-bits or adding more layers would require changing a couple values. Also, if more layers were to be added I would have needed to retrain the network and convert the weight and bias values. My weights and bias were based off my trained Pytorch network that I got to be at 100% accuracy.

**Part 1 Training the Network:**

My network consists of one linear layer and an output layer. The linear layer has 784 inputs to accommodate the 784 pixels values that get passed to it. Then the linear layer output is 64 since that seemed like a nice computer number amount. The output layer takes in the 64 outputs and will have 10 outputs where each output corresponds to the digit the networked guess. Meaning that, the highest value on index 0 means the network guessed the input digit to be a 0.

* 1. **Multi-Parameter Training**

To allow me to quickly get to the best accuracy with my Pytorch Network I utilized a python dictionary. It contained arrays for learning rate, batch size, shuffling, and epochs. The arrays had varying data types and could vary in the amount of data in the array. This method allowed me to test multiple types and get to the 100% accuracy. To get the values that would be best is mainly trial and error. So, doing multiparameter just speeds up the trial, but understanding each parameter also help.

Where the learning rate determines how far the optimization steps having it too big could result in the optimization jumping back and forth from two points as it gets closer to the min value, but too small and it might not jump far enough within the given number of epochs that were ran. When initially setting my values for the learning rate I only strayed three decimal places either way from 0.01. Ultimately, the learning rate that had the best result was 0.01. Also, how the batch size determines how many pictures are going to be loaded in, to train on. Generally, the more the better. Next is shuffling, where shuffling just determines where the pictures will be shuffled each time. This does mean that for the multiparameter that shuffle is either true or false. Finally, the number of epochs indicates the number of times that pytorch will correct the Network.

* 1. **Utilizing the GPU**

My laptop has a Nvidia GPU which is cuda capable, and pytorch makes using cuda cores ridiculously easy. By simply setting the torch.device to cuda:0 indicating the first GPU on the system, and then using .to(the set device) everything was loaded to the GPU. Using the GPU allows for larger batch sizes to be executed way faster than can be done on the CPU. Which allowed me to reach the 100% accuracy faster by using big batch sizes. If I was not going to use large batch sizes using the GPU could actually be slower since there is some delay from passing the data from the CPU to the GPU.

* 1. **Outputting to JSON File**

Inside my training code I kept track of the accuracy and loss of each epoch. Whenever a better accuracy and loss appears I would save the weights and biases to the corresponding json element. After all epochs are ran the code will write the json data to best\_results.json and add a new entry into history\_best\_results.json. The history was to be used as a reference for what parameters have gotten me the best results. The weights and biases are saved as floating point in those files to be parsed later.

**Part 2 Converting to Signed Fixed Point:**

1. **Parsing Best Results**

For converting the floating point to fixed point I load the data from best\_resuls.json file looping through the weight and bias arrays passing the value through my converter to a new json variable. Then output each weight array to its correspond dat file since the weight is a 2D array where each first index corresponds to the node the weights belong to. For the bias I just put them in one file and will have each node receive the correct bias index.

My converter does signed fixed point conversion with Q17.14 in Qm.n notation where m is the number of integer bits and n is the number fractional bits, and I am using a 32-bit address. Initially I was thinking that there did not need to be many bits for m, but then I started thinking about how the final values are calculated. Where all the input pixels are times by a weight value and then added together. Doing worst case that doesn’t really make since to do since an entire white picture is not going to happen, and only a few weights ever reached the highest value of 10. So, the worst case would be that all the pixels have a value of 1 and the weights all have a value of 10. Leaving 784 times 10 + 1 on the first layer adding one for bias. Which is then passed to 64 other nodes where each weight is also 10 and we add 1 for the bias resulting in a value of 5,018,241 for the worse case integer value. The smallest observed fractional value was .0001010… Meeting the smallest fractional was the standard I used since the worst case for the integer values did not seem like it would even happen and 2-14 yielded .00006103515625. I then gave the rest of the bits minus one for sign to the integer part. Resulting in a max in value of 131,071. The converter

1. **Parsing Pixel values**

For getting image files I utilized the dataloader from the training. I passed in a large batch and then loop through until I get a 0 target label then a 1 and so on ending at 10. Each time the target label matches the current digit I want the code will loop through the image passing every value through my converter then to the dat file for that image target. I only get one image file for each digit but it is randomized. I can also rerun the parse\_image\_data.py file to get new image values without needing to change anything in Verilog.

**Part 3 Verilog Modules:**

1. **Half Adder**
2. **Fuller Adder**
3. **Carry Lookahead Adder**
4. **Two’s Compliment Multiplier**
5. **ROMs**
6. **Control**
7. **Nodes**
8. **Layers**
9. **SoftMax**
10. **Network**

**Part 4 Connecting the Network:**

All the modules and connections are generated through python. I created multiple class for unique modules that a separate python file can access to implement each module. Then writing all the modules to a sv file is done.

1. **Initial Attempt**

I originally was not using a clk or layer enable signals. To do this meant that I was creating over 784 adders and multipliers for each node in first layer, and over 64 in the second. The output sv file was over 300,000 lines and ModelSim gave a warning saying that I needed 24448 Mbytes of RAM in order to run my design. ModelSim did crash after the warning giving memory allocation errors.

Graphical user interface, text, application, email

Description automatically generated

Figure X: ModelSim Memory Warning Message

1. **Optimization**

After the RAM issue happened, I went about trying to figure out how to optimize the code. First thing I needed to do was to just use one adder and multiplier for each node. To do that I utilized a control module which handled the clk and used always blocks inside the nodes to give input values and output values at different points in the clk cycle to the adder and multiplier. The next issue was having both layers running at the same time. The first layer needs to be complete before the next layer starts. So, I added in layer enable signals to the control module which prevented layers from running when not enabled. The enable signals are done based on keeping track of the clock cycles where a raise and fall is one cycle. The first layer requires 784 cycles to finish, and the second requires 784 + 64 cycles. The multiplying is done on the posedge of the clk and the adding is done on the negedge. The relu operation is perform on the negedge of the enable signal. The adding also feeds into its self for getting the summation, and takes the result from multiplying the input by the weight.

**Part 5 Completed Network Results:**

1. **![Chart

   Description automatically generated]()![A screenshot of a computer

   Description automatically generated with medium confidence]()Output Waves with Passed in Digit Picture**
2. **Training the Network with PyTorch:**
   * 1. Setup Network to train on using an input layer outputs 784 values, a single hidden linear layer outputs 64 values, and an output linear layer outputs 10 values.
     2. Implemented Training on the GPU, and used varying learning rates, batch sizes, to shuffle, and epochs all of which improved the speed at which I could get better results.
     3. Output the results to a json file called best\_results.json. I even made a history\_best\_results.json for keeping track of the previous best results. That was done so I could the parameters that got me to the best result the quickest.
3. **Results from Training:**
   * 1. The input layer is required to output 784 values since that is the amount of pixels we will be processing in from the 28x28 grayscale image.
     2. The single linear layer will take in those 784 values into each of the 64 nodes to output the 64 values. So that means each node has 784 weights and there are 64 biases one for each node. 64 just seemed like a nice value to output, and it was recommended by [sentdex](https://pythonprogramming.net/introduction-deep-learning-neural-network-pytorch/) to use a power of 2 value.
     3. The output layer is required to output 10 values unless I do a softmax function. Those values correspond to the guess value, and since we are trying to determine values 0-9, we have 10. The highest of the 10 is the guessed value of the network.
     4. As of writing this report my accuracy is 100% with a loss of 0.000555.
     5. This was done with the parameters setup as followed: learning rate = 0.01; batch size = 1250; to shuffle = true; and epoch reaching 49.
4. **Fixed Point Conversion:**
   * 1. I made a fixed-point convertor that takes in the desired address size, the value, the desired size of the integer part, and the desired size of the fractional part.
     2. Implemented 2’s complement on the values negative values.
     3. The converter returns the value as a string
     4. Inside Python-Parsing I setup the main.py script to loop through the best\_results.json and I initially outputted the results to their own file, but switch to just outputting to a single json.
5. **Results from Fixed Point Conversion:**
   * 1. Using Qm.n notation I did Q17.14. These values were chosen for the 32-bit address.
     2. Determining the fractional size, I did not see a decimal value smaller than 0.00010. Which means 2 -13 results in .00012 being the smallest decimal so I went to 2-14 allowing for the smallest decimal value.
     3. Determining the integer size, I needed to look at the possible values that through multiplying and adding I could get. So, I did a worse case guess where I saw the highest integer value as 9, and the bias never reached 1. Meaning the worst case biggest int value could be 784\*9\*64\*9 = 4,064,256. This is incredibly unlikely though since there really wasn’t many values being 9 and it would require every pixel to have a value of 1. So, I determined the best size for the fractional than used the rest for the sign bit and the integer parts. (Just in case this produces high error I plan on upscaling to 64 bits once I get everything implemented in order to have better accuracy)
     4. The conversion was done by multiplying the value by 2n or in my case 214 then rounding the number down to an integer value to convert it to binary. Then applying 2’s complement on the binary value if it is negative.
6. **Basic Verilog Circuits Created for Testing:**
   * 1. Half Adder
     2. 1-bit Full Adder
     3. 4-bit Full Adder
     4. 4-bit Carry Lookahead Adder
     5. 4-bit 2C Multiplier
     6. Relu function circuit that just outputs the value passed in if it’s greater than 0, and if not just 0.
7. **Results From the Basic Verilog Circuits:**
   * 1. I design modules first by hand to get the logic then wrote out how I think the Verilog code should be (I will save the designs for the final report), and then coded the circuits and their corresponding testbenches. I then tested them in [Edit code - EDA Playground](https://www.edaplayground.com/home) which allowed for quick running and editing of the testbenches and modules. (See the following figures for the waveforms of each)
     2. These basic Circuits were used to test my Verilog coding and to verify that my future python generated Verilog code is correct. Also, having them allows me to see how I could go about programmatically creating the circuits with python.
     3. The Half Adder takes in two 1-bit inputs then XORs the values together to get the sum and ANDs the values to get the carry.
     4. The 1-bit Full Adder takes in three 1-bit inputs which includes a carry in. Then performs AND, OR, and XOR operations to get the sum and carry.
     5. The 4-bit Full Adder takes in two 4-bit values and a 1-bit carry in. It is a combination of four 1-bit Full Adders with the carry outs connected to the next, and each taking in different index of the of the passed in value.
     6. The 4-bit Carry Lookahead Adder takes in a 1-bit carry in and two 4-bit values. It creates the propagation by XORing the values, and the generate by ANDing the values. The propagate and generate and then used in a combination of AND and OR gates making the carry values. The carry values are then XORed with the propagate value to get the sum.
     7. The 4-bit 2C Multiplier takes in two 4-bit values and will output a 4-bit value. The output is a 4-bit value since I chopped the actual result. The chop will vary later based on the (Qm.n) m and n values where I will chop off from the left of the address (m+1)-bits and from the right I will chop off n-bits. The multiplier is a large collection of ANDs, NANDs, 1-bit Full Adders, and Half adders.

![A screenshot of a computer

Description automatically generated]()

*Figure 1: Half Adder Waveform*

![A screenshot of a computer

Description automatically generated]()

*Figure 2: 1-Bit Full Adder Waveform*

*![A computer screen capture

Description automatically generated with medium confidence]()*

*Figure 3: 4-Bit Full Adder Waveform*

*![Graphical user interface

Description automatically generated]()*

*Figure 4: 4-bit Carry Lookahead Adder Waveform*

![Graphical user interface

Description automatically generated with medium confidence]()

*Figure 5: 4-bit 2’s Complement Multiplier Waveform*

**Part 2 What is Left**:

1. **Generate n-Bit Verilog Modules with Python:**
2. I will create a python script that will generate a desired bit size and outputs the module.
3. **Build Network with the Verilog Modules:**
4. Create a testbench using python to parse through the best\_results and hook up the modules creating the network.
5. Possible implement the softmax for outputting the number the network thinks the image is rather than just looking at the values and picking which one is the highest.
6. **Output Test Images to Files:**
7. Create various files to read into the network in the testbench. I will output the pixel values to a file one value per line and then read the values in with the Verilog testbench.
8. **Implement CLA in Multiplier:**
9. I will need to make a 1-bit, 2-bit, and 3-bit CLA as well. Since the CLA shouldn’t go over the 4-bit design.
10. I will need to modify by removing the full adders and replacing them with the a combination of n-bit CLAs where n is less than or equal to 4.
11. **Implement Dadda Tree Multiplier: (Optional / If there is time)**
12. The dadda multiplier would be much faster and should save power than the basic 2’s complement multiplier that I am doing.
13. **Upscale Address to 64-Bit: (Optional / If there is time)**
14. Will improve accuracy of fixed values resulting in a more accurate final guess
15. **Create Additional Layers: (Optional / If there is time)**
16. Implementing additional layers would improve the accuracy of detecting the numbers, but it would require me to rebuild the network which could take some time to do. That is why I am saving this for last.