Midterm Report

Timothy Baalman

ECEN 4303

<u>Part 1 What I Have Completed</u>: (All Code is in the attached zip file)

1) Training the Network with PyTorch:

i. 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.

ii. 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.

iii. 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.

2) Results from Training:

i. 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.

ii. 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 to use a power of 2 value.

iii. 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.

iv. As of writing this report my accuracy is 100% with a loss of 0.000555.

v. This was done with the parameters setup as followed: learning rate = 0.01; batch size = 1250; to shuffle = true; and epoch reaching 49.

3) Fixed Point Conversion:

i. 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.

ii. Implemented 2's complement on the values negative values.

iii. The converter returns the value as a string

iv. 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.

4) Results from Fixed Point Conversion:

i. Using Qm.n notation I did Q17.14. These values were chosen for the 32-bit address.

ii. 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.

iii. 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)

iv. The conversion was done by multiplying the value by 2^n or in my case 2^{14} 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.

5) Basic Verilog Circuits Created for Testing:

i. Half Adder
ii. 1-bit Full Adder
iii. 4-bit Full Adder
iv. 4-bit Carry Lookahead Adder
v. 4-bit 2C Multiplier
vi. Relu function circuit that just outputs the value passed in if it's greater than 0, and if not just 0.

6) Results From the Basic Verilog Circuits:

i. 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 which allowed for quick running and editing of the testbenches and modules. (See the following figures for the waveforms of each)

ii. 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.

iii. 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.

iv. 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.

v. 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.

vi. 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.

vii. 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.

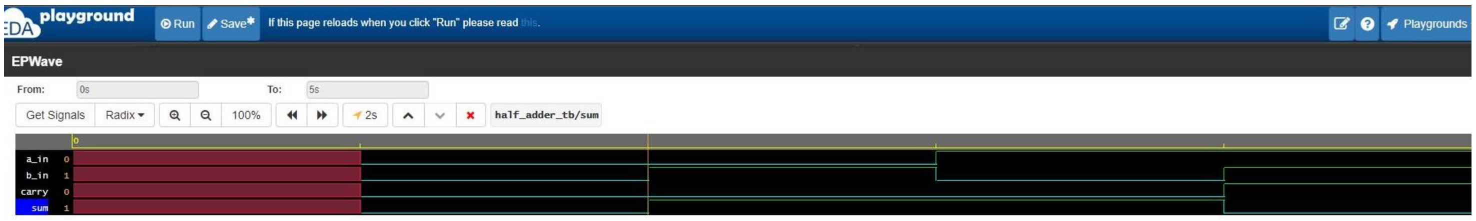

Figure 1: Half Adder Waveform

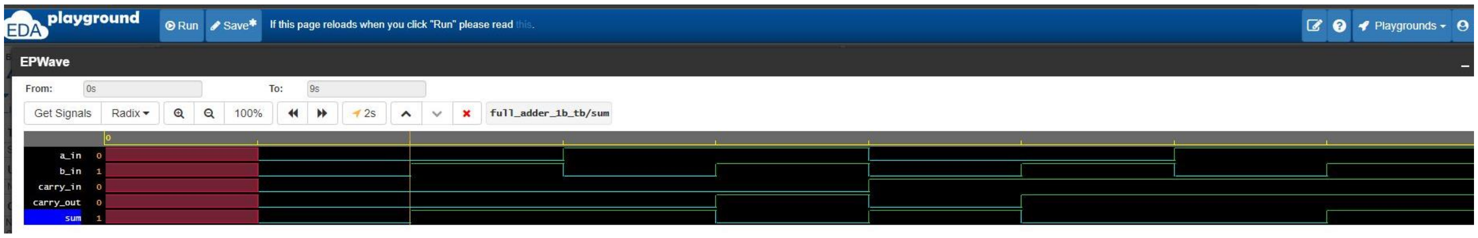

Figure 2: 1-Bit Full Adder Waveform

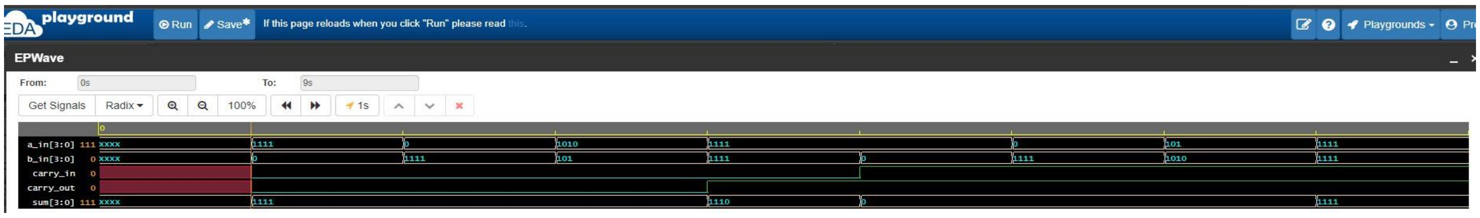

Figure 3: 4-Bit Full Adder Waveform

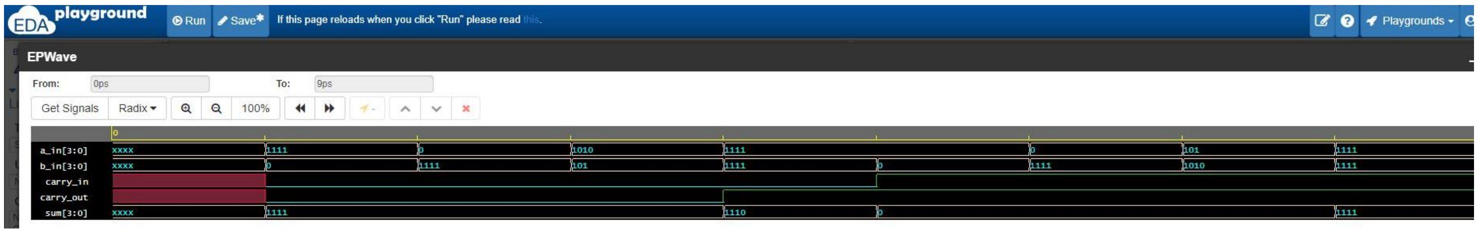

Figure 4: 4-bit Carry Lookahead Adder Waveform

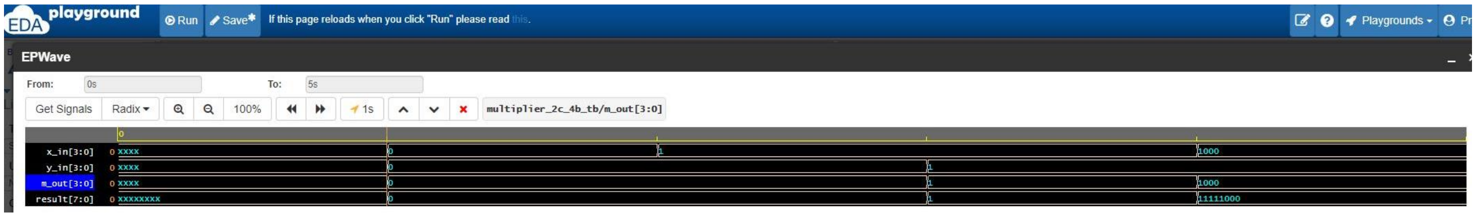

Figure 5: 4-bit 2's Complement Multiplier Waveform

Part 2 What is Left:

1) **Generate n-Bit Verilog Modules with Python:**
   i. I will create a python script that will generate a desired bit size and outputs the module.

2) **Build Network with the Verilog Modules:**
   i. Create a testbench using python to parse through the best_results and hook up the modules creating the network.
   ii. 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.

3) **Output Test Images to Files:**
   i. 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.

4) **Implement CLA in Multiplier:**
   i. 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.
   ii. 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.

5) **Implement Dadda Tree Multiplier: (Optional / If there is time)**
   i. The dadda multiplier would be much faster and should save power than the basic 2's complement multiplier that I am doing.

6) **Upscale Address to 64-Bit: (Optional / If there is time)**
   i. Will improve accuracy of fixed values resulting in a more accurate final guess

7) **Create Additional Layers: (Optional / If there is time)**
   i. 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.