FPGA Neural Net Implementation

An Independent Study Report

Sarah Brown
College of Engineering
University of Oklahoma
Norman, OK

***Abstract*— A FPGA implementation of a simple neural network to identify numbers trained on the MNIST dataset. By taking the weights and biases of a trained model, it is possible to reconstruct the model for inference purposes on a FPGA. This allows for the model to be run very quickly and independently.**

***Keywords*—FPGA, Neural Net, MNIST Dataset, Inference**

I. Introduction

A. *Artificial Neural Networks*

Neural networks are a type of machine learning modeled after the neurons of a brain. They are very effective at creating models for tasks that can be very difficult for computers to do traditionally. Any machine learning task can be broken into two parts, training and inference. Neural networks work by summing together various inputs with different weights. Training helps to determine the different weights and biases that result in the best accuracy for a model. Training has been studied in depth to effectively use software to create optimal models, but once a model is trained it simply needs to be run through the created architecture. Due to the effectiveness and potential for neural networks, there is a large amount of interest in creating specialized hardware to run inputs through the constructed neural network model to produce outputs.

B. *FPGA Background*

There are various different ways of implementing hardware solutions to perform inference on inputs. Application-specific integrated circuits (ASIC) can be designed to perform specific operations. One example is TPU Cloud, designed by Google to accelerate machine learning operations [1]. However, ASICs can take a long time to develop. Field-programmable gate arrays (FPGAs) are a type of integrated circuit that can be configured with a hardware description language (HDL). As FPGAs can be reconfigured, they are ideal for testing and development.

C. *Use of FPGAs for Inference*

While they are less efficient due to being reconfigurable, FPGAs are much more flexible than ASICs for development. This creates an area of research in the middle between the two ends of efficiency and flexibility. [3] One example of this middle-ground is a project developed by Microsoft, Project Brainwave. Project Brainwave works to run pre-trained neural networks on FPGAs to increase the efficiency of Microsoft datacenters. FPGAs allow for rapid development and can be reprogrammed as AI algorithms are improved and updated.

D. *Reason for Interest*

Both FPGAs and neural networks are large and expanding areas of study. Due to increased interest in these fields, the skill sets gained from working on a project related to both areas can be beneficial on a professional level. In addition to reviewing FPGA knowledge learned in previous course, this project allows further expansion into overlap between disciplines.

The Hello, World of neural networks is to be able to identify a handwritten digit from the MNIST dataset. This is a solved problem and one that can be implemented with a simple neural network. This project creates a neural network architecture in Python with the Keras package and then implements it on a FPGA with a HDL, Verilog.

II. Python

A. *Keras Model*

With the use of the Keras package, it is trivial to instantiate a model and then train it using the MNIST dataset. The model architecture is a fully-connected neural network. The inputs are all fed into the first layer neurons via a weighted sum and the first layer is then connected to the second layer. The maximum value from the second layer is then selected to represent the output. Between the layers, an activation function is used to determine when the neuron is activated. There are many different types of activation functions. One of these is the rectified linear unit (ReLU) activation function. The ReLU function is a piecewise linear function that is 0 for negative numbers and linear with a slope of 1 for positive numbers.

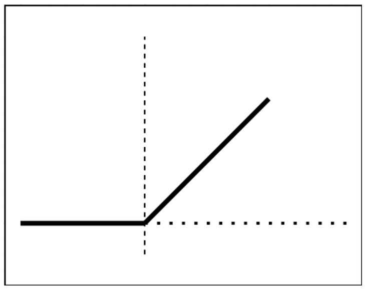
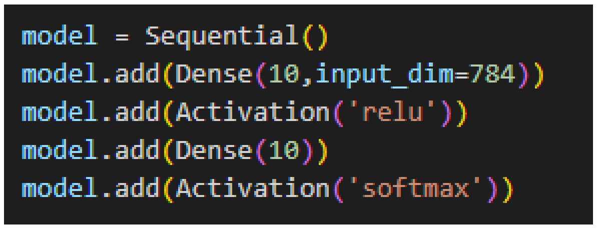


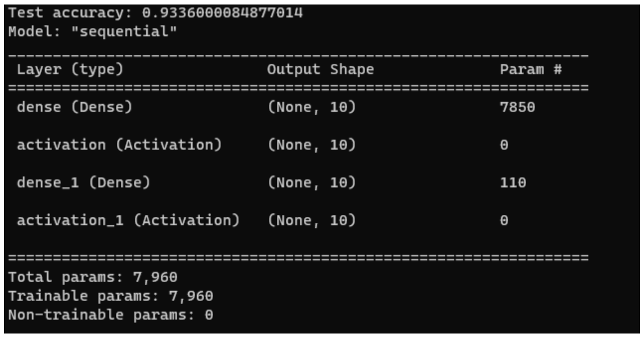
Image 1: ReLU Function.

Due the linearity of this activation function, it makes the hardware implementation much easier. During training, the weights and biases per layer are adjusted to increase model accuracy. After training, these values can be reconstructed and used to recreate the model architecture for deployment.



```
model = Sequential()
model.add(Dense(10,input_dim=784))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

Image 2: Model instantiation with Keras package.



```
Test accuracy: 0.9336000084877014
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                7850

 activation (Activation)     (None, 10)                0

 dense_1 (Dense)             (None, 10)                110

 activation_1 (Activation)   (None, 10)                0

=================================================================
Total params: 7,960
Trainable params: 7,960
Non-trainable params: 0
```

Image 3: Model details and testing accuracy.

B. *Python Reimplementation*

Before transitioning the model architecture to hardware, the model was then recreated in Python without the Keras package. By taking the saved weights and biases, it was possible to loop over the test images and confirm that the testing accuracy remained consistent. This reimplementation also allowed for value confirmation during FPGA simulation, proving valuable for debugging Verilog code issues.

C. *Floating Point to Fixed Point*

With confirmation of a working architecture, the next step is to output the weights and biases in a way that can be imported to FPGA memory. However, before writing these values to memory initialization files, the floating-point numbers were converted to fixed-point numbers. Fixed-point numbers are easier to understand and implement in the context of hardware. There is a small loss of precision when converting to fixed-point, but the advantages far outweigh the resulting small differences. In addition, conversion to and from floating point was implemented in the pure python version to confirm that this accuracy difference would not impact the result.

III. FPGA

A. *FPGA Code*

With the saved model values in fixed-point format, the next step was to try to implement a first layer neuron and verify its expected output versus its actual output using the step-by-step values from the pure python implementation. The first layer neuron reads in the bias values and then iterates over each pixel to sum the weighted values. The ReLU function is then applied, resulting in a completed neuron.

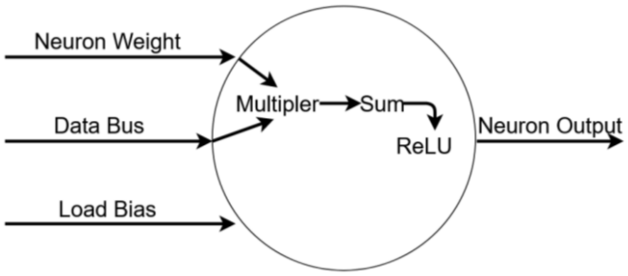
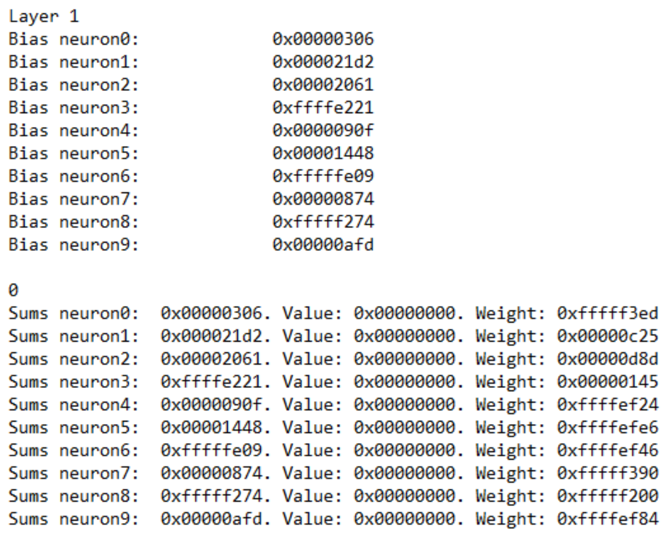


Image 4: Layer 1 neuron.

Using ModelSim, a FPGA simulation tool, neuron 0 of the first layer was tested by loading in the pixel values for the first test image from the MNIST dataset. The expected values, the pixel 0 values for all 10 neurons are shown below, were compared to the simulation results to confirm accuracte FPGA implementation. A ModelSim screenshot can be seen in Appendix B image 2.



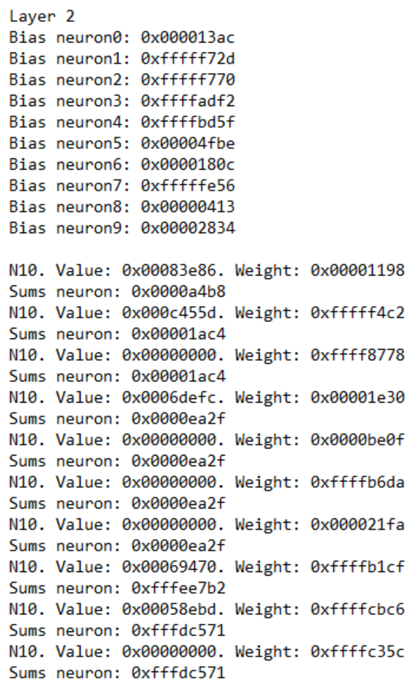
```
Layer 1
Bias neuron0:              0x00000306
Bias neuron1:              0x000021d2
Bias neuron2:              0x00002061
Bias neuron3:              0xffffe221
Bias neuron4:              0x0000090f
Bias neuron5:              0x00001448
Bias neuron6:              0xfffffe09
Bias neuron7:              0x00000874
Bias neuron8:              0xfffff274
Bias neuron9:              0x00000afd

0
Sums neuron0:  0x00000306. Value: 0x00000000. Weight: 0xfffff3ed
Sums neuron1:  0x000021d2. Value: 0x00000000. Weight: 0x00000c25
Sums neuron2:  0x00002061. Value: 0x00000000. Weight: 0x00000d8d
Sums neuron3:  0xffffe221. Value: 0x00000000. Weight: 0x00000145
Sums neuron4:  0x0000090f. Value: 0x00000000. Weight: 0xffffef24
Sums neuron5:  0x00001448. Value: 0x00000000. Weight: 0xffffefe6
Sums neuron6:  0xfffffe09. Value: 0x00000000. Weight: 0xffffef46
Sums neuron7:  0x00000874. Value: 0x00000000. Weight: 0xfffff390
Sums neuron8:  0xfffff274. Value: 0x00000000. Weight: 0xfffff200
Sums neuron9:  0x00000afd. Value: 0x00000000. Weight: 0xffffef84
```

Image 5: Values for layer 1 pixel 0 calculated from pure python method.

After confirmation that neuron 0 worked as intended, copies of this module were instantiated to form all ten neurons of layer 1.

Similarly, the first neuron of layer 2 was created and tested. Instead of iterating across each pixel as was done in layer 1, the neurons of layer 2 receive each of the output values from layer 1. This was again confirmed to function as intended via simulation.



```
Layer 2
Bias neuron0: 0x000013ac
Bias neuron1: 0xfffff72d
Bias neuron2: 0xfffff770
Bias neuron3: 0xffffadf2
Bias neuron4: 0xffffbd5f
Bias neuron5: 0x00004fbe
Bias neuron6: 0x0000180c
Bias neuron7: 0xfffffe56
Bias neuron8: 0x00000413
Bias neuron9: 0x00002834

N10. Value: 0x00083e86. Weight: 0x00001198
Sums neuron: 0x0000a4b8
N10. Value: 0x000c455d. Weight: 0xfffff4c2
Sums neuron: 0x00001ac4
N10. Value: 0x00000000. Weight: 0xffff8778
Sums neuron: 0x00001ac4
N10. Value: 0x0006defc. Weight: 0x00001e30
Sums neuron: 0x0000ea2f
N10. Value: 0x00000000. Weight: 0x0000be0f
Sums neuron: 0x0000ea2f
N10. Value: 0x00000000. Weight: 0xffffb6da
Sums neuron: 0x0000ea2f
N10. Value: 0x00000000. Weight: 0x000021fa
Sums neuron: 0x0000ea2f
N10. Value: 0x00069470. Weight: 0xffffb1cf
Sums neuron: 0xfffee7b2
N10. Value: 0x00058ebd. Weight: 0xffffcbc6
Sums neuron: 0xfffdc571
N10. Value: 0x00000000. Weight: 0xffffc35c
Sums neuron: 0xfffdc571
```

Image 6: Values for neuron 10 layer 2, calculated from pure python method

However, no ReLU function is applied to these neurons and instead a softmax is applied to all of layer 2 to select the output with the highest value. This value is then fed through a seven-segment display decoder and shown on the FPGA board.

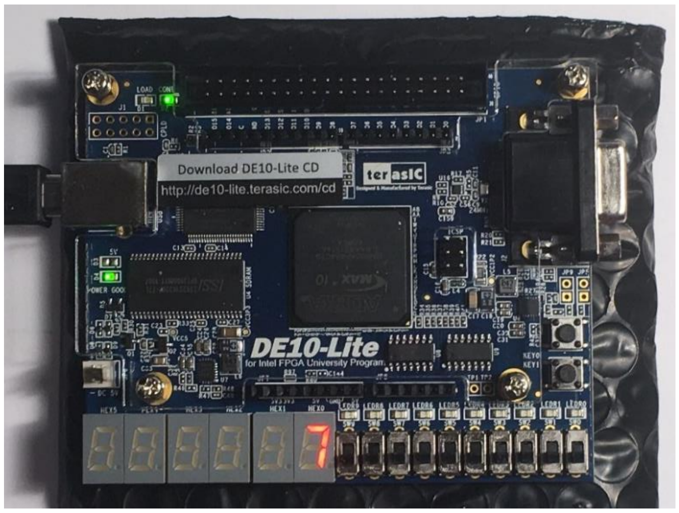

Download DE10-Lite CD
http://de10-lite.terasic.com/cd
terasIC
DE10-Lite
for Intel FPGA University Program

Image 7: Display on DE10-Lite showing the correctly identified 7.

To control the flow of data between the network's layers, a state machine was implemented as a control block. This control block ensures that the memory is loaded correctly and switches states to process the loaded pixel values. The full schematic of the resulting from the FPGA code can be viewed in Appendix B image 4.

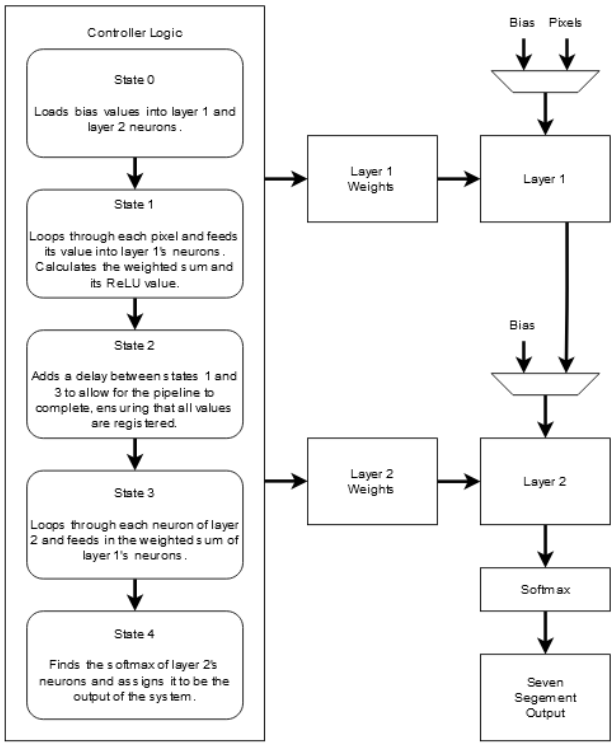

Controller Logic
State 0
Loads bias values into layer 1 and layer 2 neurons.
State 1
Loops through each pixel and feeds its value into layer 1's neurons. Calculates the weighted sum and its ReLU value.
State 2
Adds a delay between states 1 and 3 to allow for the pipeline to complete, ensuring that all values are registered.
State 3
Loops through each neuron of layer 2 and feeds in the weighted sum of layer 1's neurons.
State 4
Finds the softmax of layer 2's neurons and assigns it to be the output of the system.
Bias
Pixels
Layer 1 Weights
Layer 1
Bias
Layer 2 Weights
Layer 2
Softmax
Seven Segement Output

Image 8: FPGA block diagram.

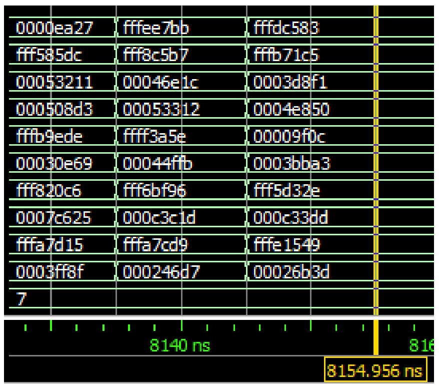
IV. Conclusion

A. Results

The most obvious result is the fact that the first test image was correctly identified and displayed as a seven. However, further comparison can be done via the time it took for each operation. The FPGA implementation resulted in the fastest time to predict an image based on a preconstructed model by over a factor of 8. Additional streamlining of the FPGA code could further increase this time gap.

	Time
Pure Python	186000 µs
Keras	68.5 µs
FPGA	8.1 µs

Table 1: Time results.



```
0000ea27  fffee7bb  fffdc583
fff585dc  fff8c5b7  fffb71c5
00053211  00046e1c  0003d8f1
000508d3  00053312  0004e850
fffb9ede  ffff3a5e  00009f0c
00030e69  00044ffb  0003bba3
fff820c6  fff6bf96  fff5d32e
0007c625  000c3c1d  000c33dd
fffa7d15  fffa7cd9  fffe1549
0003ff8f  000246d7  00026b3d
7
```

Image 9: Time result for FPGA implementation.

B. Further Work

Additional work on this project could be done to further decrease the time required to predict an image. One way to do this would be to increase the neurons in layer 1 and split the image in half between the increased number of neurons. In addition, currently only one image is stored in memory, but work could be done to feed images in to predict over a high-speed interface. As a final step, a camera could be added to the FPGA setup to predict handwritten digits in real time.

References

[1] "Cloud TPU," Google. [Online]. Available: https://cloud.google.com/tpu. [Accessed: 09-May-2022].

[2] "Project Brainwave," Microsoft Research, 12-Nov-2020. [Online]. Available: https://www.microsoft.com/en-us/research/project/project-brainwave/. [Accessed: 26-Apr-2022].

[3] "FPGA based acceleration of machine learning algorithms involving convolutional neural networks," RSS, 01-Jul-2020. [Online]. Available: https://thedatabus.io/introduction. [Accessed: 26-Apr-2022].

[4] "Verilog For Loop," ChipVerify. [Online]. Available: https://www.chipverify.com/verilog/verilog-for-loop. [Accessed: 08-May-2022].

[5] J. Brownlee, "A gentle introduction to the rectified linear unit (ReLU)," Machine Learning Mastery, 20-Aug-2020. [Online]. Available:

[6] V. Kizheppatt, "Neural Networks on FPGA: Part 1: Introduction - YouTube," YouTube. [Online]. Available: https://www.youtube.com/watch?v=rw_JITpbh3k. [Accessed: 09-May-2022].

[7] Intel FPGA, "Machine Learning on FPGAs: Neural Networks - YouTube," YouTube. [Online]. Available: https://www.youtube.com/watch?v=3iCifD8gZ0Q. [Accessed: 09-May-2022].

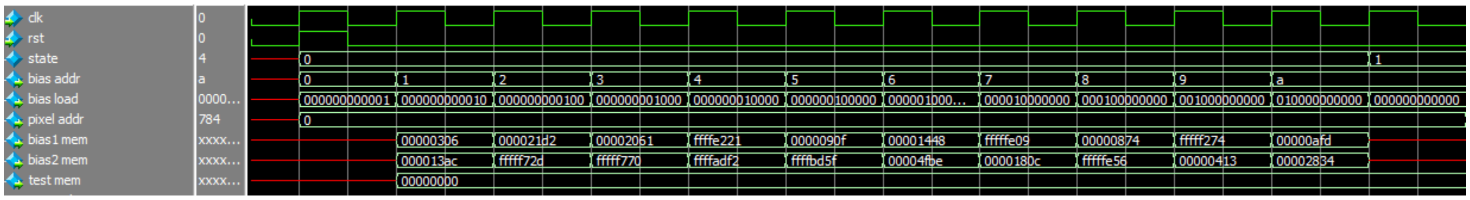
[8] 3Blue1Brown, "But what is a neural network? | Chapter 1, Deep Learning – YouTube." YouTube [Online]. Available: https://www.youtube.com/watch?v=aircAruvnKk. [Accessed: 09-May-2022].

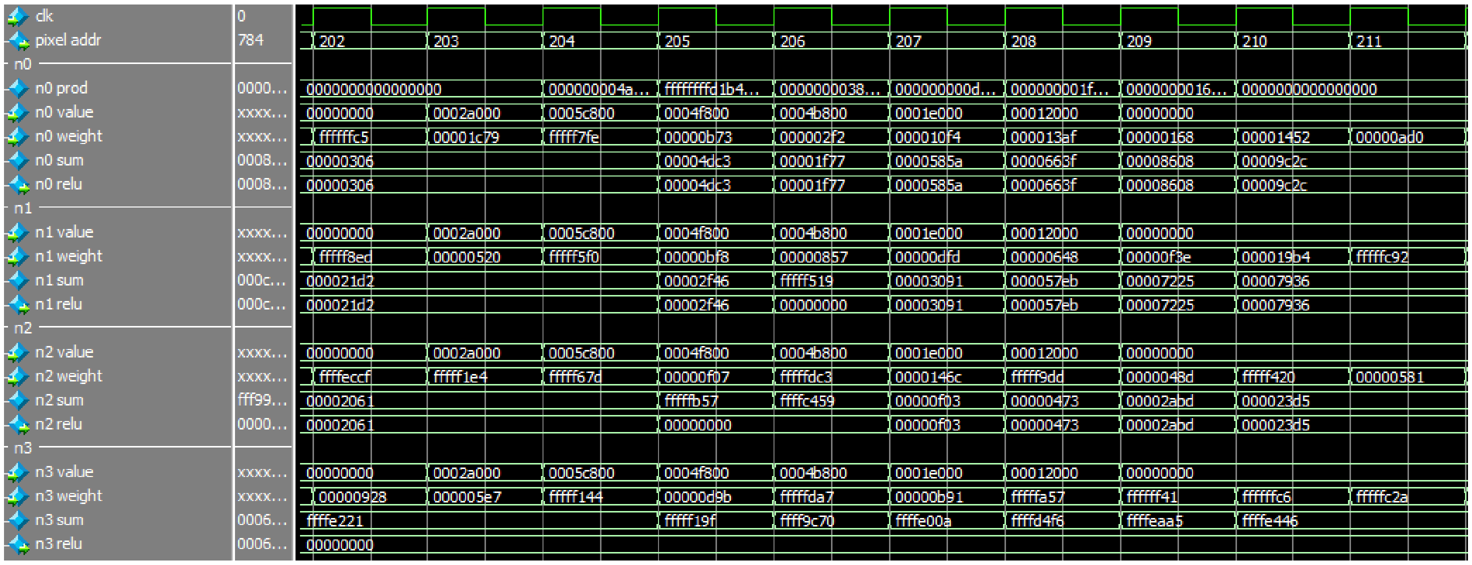
[9] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/. [Accessed: 08-May-2022].

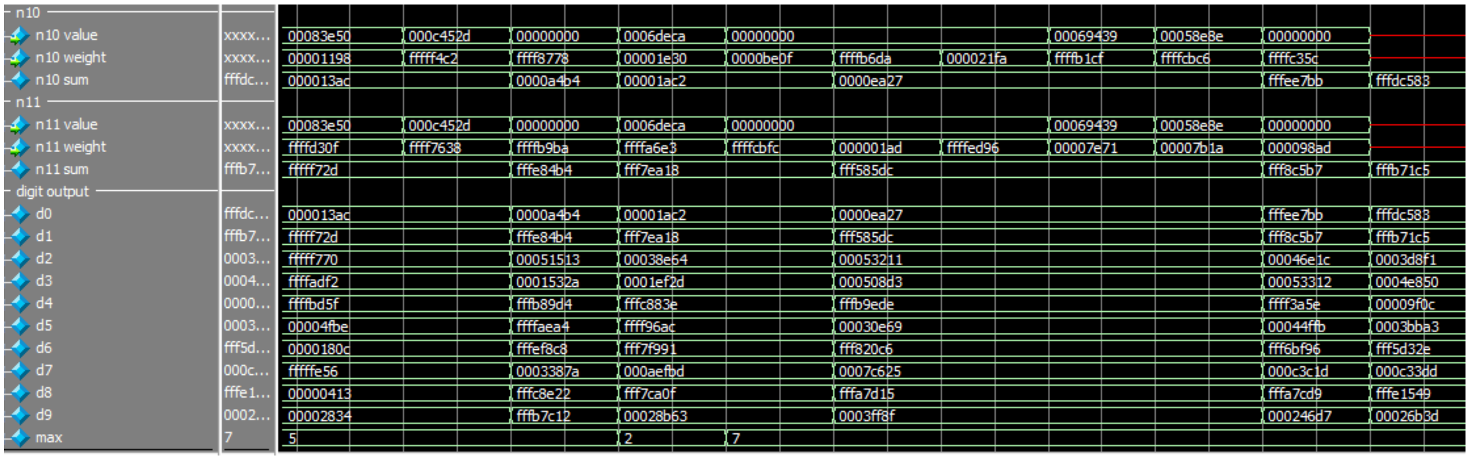
APPENDIX A: CODE

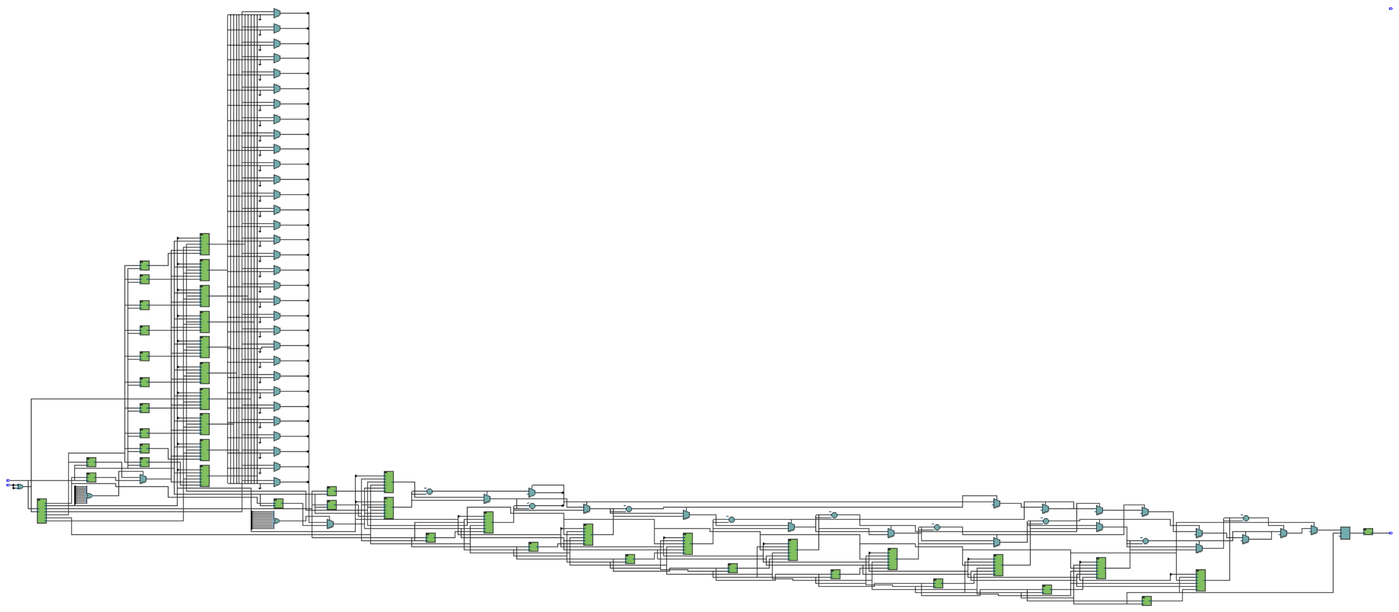
The Python and Verilog code for this project can be viewed on my Github at https://github.com/SarahBrown/fpga-mnist-dataset

APPENDIX B. IMAGES


clk 0
rst 0
state 4 0 1
bias addr a 0 1 2 3 4 5 6 7 8 9 a
bias load 0000... 000000000001 000000000010 000000000100 000000001000 000000010000 000000100000 000001000... 000010000000 000100000000 001000000000 010000000000 000000000000
pixel addr 784 0
bias1 mem xxxx... 00000306 000021d2 00002061 ffffe221 0000090f 00001448 fffffe09 00000874 ffff274 00000afd
bias2 mem xxxx... 000013ac fffff72d fffff770 ffffadf2 ffffbd5f 00004fbe 0000180c fffffe56 00000413 00002834
test mem xxxx... 00000000
Appendix B: Image 1. Confirmation of bias values.


clk 0
pixel addr 784 202 203 204 205 206 207 208 209 210 211
n0
n0 prod 0000... 0000000000000000 000000004a... ffffffffd1b4... 0000000038... 000000000d... 000000001f... 0000000016... 0000000000000000
n0 value xxxx... 00000000 0002a000 0005c800 0004f800 0004b800 0001e000 00012000 00000000
n0 weight xxxx... ffffffc5 00001c79 fffff7fe 00000b73 000002f2 000010f4 000013af 00000168 00001452 00000ad0
n0 sum 0008... 00000306 00004dc3 00001f77 0000585a 0000663f 00008608 00009c2c
n0 relu 0008... 00000306 00004dc3 00001f77 0000585a 0000663f 00008608 00009c2c
n1
n1 value xxxx... 00000000 0002a000 0005c800 0004f800 0004b800 0001e000 00012000 00000000
n1 weight xxxx... fffff8ed 00000520 fffff5f0 00000bf8 00000857 00000dfd 00000648 00000f3e 000019b4 fffffc92
n1 sum 000c... 000021d2 00002f46 fffff519 00003091 000057eb 00007225 00007936
n1 relu 000c... 000021d2 00002f46 00000000 00003091 000057eb 00007225 00007936
n2
n2 value xxxx... 00000000 0002a000 0005c800 0004f800 0004b800 0001e000 00012000 00000000
n2 weight xxxx... ffffeccf fffff1e4 ffff67d 00000f07 fffffdc3 0000146c ffff9dd 0000048d fffff420 00000581
n2 sum fff99... 00002061 fffffb57 ffffc459 00000f03 00000473 00002abd 000023d5
n2 relu 0000... 00002061 00000000 00000f03 00000473 00002abd 000023d5
n3
n3 value xxxx... 00000000 0002a000 0005c800 0004f800 0004b800 0001e000 00012000 00000000
n3 weight xxxx... 00000928 000005e7 fffff144 00000d9b fffffda7 00000b91 fffffa57 ffffff41 ffffffc6 fffffc2a
n3 sum 0006... ffffe221 fffff19f ffff9c70 ffffe00a ffffd4f6 ffffeaa5 ffffe446
n3 relu 0006... 00000000
Appendix B: Image 2. Layer 1pixel values.


n10
n10 value xxxx... 00083e50 000c452d 00000000 0006deca 00000000 00069439 00058e8e 00000000
n10 weight xxxx... 00001198 fffff4c2 ffff8778 00001e30 0000be0f ffffb6da 000021fa ffffb1cf ffffcbc6 ffffc35c
n10 sum fffdc... 000013ac 0000a4b4 00001ac2 0000ea27 fffee7bb fffdc583
n11
n11 value xxxx... 00083e50 000c452d 00000000 0006deca 00000000 00069439 00058e8e 00000000
n11 weight xxxx... ffffd30f ffff7638 ffffb9ba ffffa6e3 ffffcbfc 000001ad ffffed96 00007e71 00007b1a 000098ad
n11 sum ffb7... fffff72d fffe84b4 fff7ea18 fff585dc fff8c5b7 fffb71c5
digit output
d0 fffdc... 000013ac 0000a4b4 00001ac2 0000ea27 fffee7bb fffdc583
d1 fffb7... fffff72d fffe84b4 fff7ea18 fff585dc fff8c5b7 fffb71c5
d2 0003... fffff770 00051513 00038e64 00053211 00046e1c 0003d8f1
d3 0004... ffffadf2 0001532a 0001ef2d 000508d3 00053312 0004e850
d4 0000... ffffbd5f fffb89d4 fffc883e fffb9ede ffff3a5e 00009f0c
d5 0003... 00004fbe ffffaea4 ffff96ac 00030e69 00044ffb 0003bba3
d6 fff5d... 0000180c fffef8c8 fff7f991 fff820c6 fff6bf96 fff5d32e
d7 000c... fffffe56 0003387a 000aefbd 0007c625 000c3c1d 000c33dd
d8 fffe1... 00000413 fffc8e22 fff7ca0f fffa7d15 fffa7cd9 fffe1549
d9 0002... 00002834 fffb7c12 00028b63 0003ff8f 000246d7 00026b3d
max 7 5 2 7
Appendix B: Image 3. Neuron 10 and neuron 11 in the second layer as well as the final digit output.



Appendix B: Image 4. Schematic of the internal structure of the design netlist. A larger version of this schematic can be viewed here: https://github.com/SarahBrown/fpga-mnist-dataset/netlist.pdf