Watching Neural Networks Learn: FPGA Implementation of Artificial Learning

Joseph Volcic, George Trupiano, Dominic Abdal, Kyle Verellen

ECE 4710 Computer Hardware Design

Oakland University

Rochester, MI, USA

volcic@oakland.edu, grupiano@oakland.edu, dabdal@oakland.edu, kverellen@oakland.edu

Abstract

In this project, the design and implementation of a neural network on a field-programmable gate array (FPGA) is explored. The neural network is tailored to learn and display a 10x10 pixel image on a VGA screen. The objective is to create an efficient solution capable of recognizing and displaying pixelated images in real time. The chosen neural network architecture, based on much testing, was adapted for implementation on the Nexys A-7 board. VHDL was employed for the encoding of the neural network's layers and training algorithms onto the board. Various simulations were conducted to validate the networks learning capabilities and were compared with results from the same network structure built in python to ensure correct execution in VHDL. Results showed the creation of a successful network in python. The operation of many components in the network were validated but a failed completion of the VHDL top file caused an ultimately non-functioning final product on the FPGA board.

Table of Contents

Page

1. **Introduction** 4
2. **Background** 4-7
3. **Implementation** 7-12
4. **Results** 13- 16
5. **Conclusions** 16
6. **References** 17

1. Introduction

Neural networks are a method in AI that teaches computers to learn in a way similar to a brain. It uses nodes, called “neurons”, interconnected in a layered structure that resembles a brain. It creates a system that computers use to continuously improve by learning from their mistakes. These networks have demonstrated remarkable capabilities in pattern recognition, learning, and decision making. The ability of neural networks to adapt and learn from new data makes them versatile tools, and their deployments on specialized hardware, such as FPGAs, holds the potential to unlock new possibilities for accelerated and efficient computation.

This project focuses on the creation of a neural network designed to learn and display a 10x10 pixel image in real time on a VGA screen using an FPGA board, specifically the Nexys A-7 (a development board build around the Artix-7 FPGA). The scope of this project encompasses VHDL based programming of the neural network and subsequent simulation and validation of its learning and inference capabilities. The inherent constraints of embedded systems require intelligent solutions that balance performance and resource utilization. The constraints of this design include the limited resources of the FPGA, requiring efficient optimization and intelligent design practices to overcome.

This report is structured to provide a comprehensive understanding of the product, starting with an introduction to provide the necessary context. The subsequent sections delve into the background and planning of the project's architecture before expanding onto the design implemented. Challenges encountered and their solutions are discussed in detail before an explanation of the results of the project. A conclusion reviewing the main points given is used to close the report.

With image-based applications being prevalent in various fields, such as robotics and surveillance, the significance of efficient image processing is apparent. Through this undertaking, this project hopes to contribute to the growing field of FPGA accelerated computation while also providing a practical demonstration of Nexys A-7's potential in real time image processing applications.

2. Background

There are many techniques to approach artificial learning, all with different advantages. Choosing a suitable method for implementation on FPGAs required careful consideration of many factors. The first of which is the mathematical approach. Many people use vectorized math to speed up computation on general purpose processors, however this approach has limited benefit when designing special purpose hardware due to the ability to create highly parallel processing blocks. With this in mind, we decided to use element wise mathematical operations over the typical vectorized approach to take full advantage of programmable hardware. Then, we needed to decide on the network structure we wanted to use. We tackled this task by building a python model of our network and tested many different possible structures.

2.1. Math Behind the Network

When working with artificial learning, you can break the learning process down into two main parts, forwards propagation and backwards propagation. The forward pass computes the predicted output of the neural network based on input data. Our network relies on an x and y coordinate to predict a red, blue, and green output. The backwards pass is where the learning happens. The network uses its predicted output and compares it to the correct output, then updates its parameters to minimize the loss. Mathematically, the forwards pass of the network is a series of nonlinear functions, which are scaled, biased, and summed to create a new function that represents the output as a function of the inputs. Each of these weighted summers accompanied by a nonlinear activation function represents a neuron. In our application, we represented this process with the following two equations:

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = g^{(l+1)}(z^{(l+1)})$$

Where "z" is the weighted sum of the function from the previous layer, "W" is the weight, "a" is the activation from a previous neuron, and "b" is the bias for a neuron. In the second equation, "a" represents the activation of the current neuron and "g" is the activation function for that neuron.

After obtaining the prediction from the forwards pass, we can now adjust our network. The backward pass does this by computing the gradients of the loss with respect to the parameters using the chain rule. Loss is calculated with the following equation:

$$J = \frac{1}{2}\left((R_{\text{pred}} - R_{\text{target}})^2 + (G_{\text{pred}} - G_{\text{target}})^2 + (B_{\text{pred}} - B_{\text{target}})^2\right)$$

Where J is the loss, and the other terms are the predicted color and target color. The loss can be propagated back to the neurons in our network by taking the derivative with respect to the weighted sum of a neuron, which is equal to the derivative of the activation function. Using this technique, we obtain the following

equations for the sensitivity of output layer. Note sensitivity quantifies the impact of the neuron's output on the overall loss.
Sensitivity also needs to be calculated for neurons that come before the output layer, this can be done

$$\delta_{\text{red}}^{(L)} = \frac{\partial J}{\partial R_{\text{pred}}} = R_{\text{pred}} - R_{\text{target}}$$
$$\delta_{\text{green}}^{(L)} = \frac{\partial J}{\partial G_{\text{pred}}} = G_{\text{pred}} - G_{\text{target}}$$
$$\delta_{\text{blue}}^{(L)} = \frac{\partial J}{\partial B_{\text{pred}}} = B_{\text{pred}} - B_{\text{target}}$$

through a process similar to the forward pass. The sensitivity of non-output layer neurons can be computed by taking the weighted sum of sensitivities from the next layer and multiplying it by the derivative of the current neuron's activation function. This equation follows:

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot g^{(l)'}(z^{(l)})$$

Now that the amount a neuron affects the loss is calculated, the weights and biases can be adjusted to try and minimize the loss. This is done taking the current weight and subtracting the change we want to make. Where the change is computed by taking the derivative of loss with respect to the change in weight. The bias undergoes a similar change.

$$W^{(l)} = W^{(l)} - \alpha \frac{\partial J}{\partial W^{(l)}}$$
$$b^{(l)} = b^{(l)} - \alpha \frac{\partial J}{\partial b^{(l)}}$$

Using these formulas, the mathematical foundation for our neural network is established.

2.2. Structuring the Network

Finding a network structure that was suitable for predicting a color based on x and y coordinates was done by using a mathematically equivalent python network model and modifying parameters within the network. Namely, the number of hidden layers, number of neurons in layer, and network inputs were modified. The initial network design was composed of an x and y input, one hidden layer, and an output layer with 3 output neurons, to represent the red, green, and blue color value based on our inputs. This network performed well if we had an abundant number of hidden layer neurons and lots of training time. To reduce the number of hidden layer neurons, multiple hidden layers were added to the network. This structure reduced the total number of neurons needed. However, the training times were longer, and the network completely failed more often.

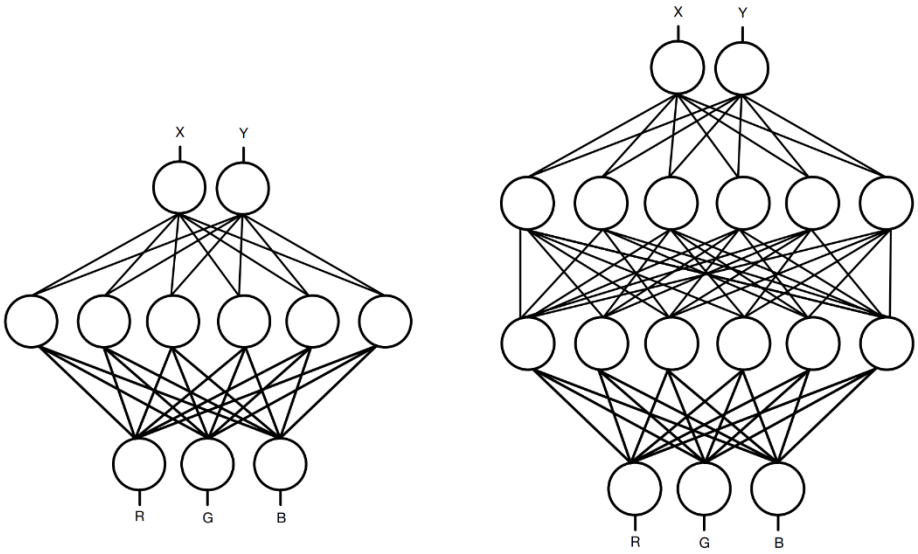

Figure 1 and 2. Network Structure with 1 and 2 Hidden Layers

After experimenting with these parameters, adding more network inputs was recommended. With this recommendation, giving the network the last correct pixel color was added as an input. This network performed exceptionally well, only requiring one hidden layer and three neurons in the hidden layer to get a good output.

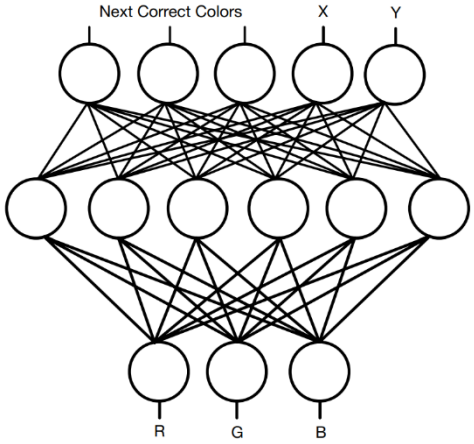

Figure 3. Network Structure with the Previous Correct RGB Value as an Input

However, the idea of giving the network a correct RGB value as an input meant we needed to know the image to predict the image. To solve this problem, instead of giving the network the last correct pixel value, the last predicted pixel was given to the network. With this model, the learning time was longer, but the benefit of the next correct pixel was still prevalent. We then tried adding the last two predicted pixels to the network. This helped further reduce the number of hidden layer neurons.

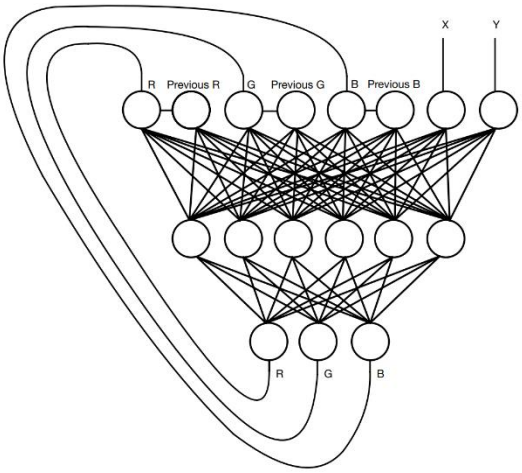

Figure 4. Network Structure Containing the Last Two Predicted RGB Values

With this network design, we then moved onto implementing the design in hardware.

3. Implementation

Now that the general structure of the Neural Network is in place, translating it to its hardware components is the next step. This was made difficult since back propagation is going to be implemented all throughout this system. To create a neural network, each component needs to be broken down into its most simple components. Each neuron consists of the following subcomponents: a weight block, weighted summer, activation function, and a backpropagation path.

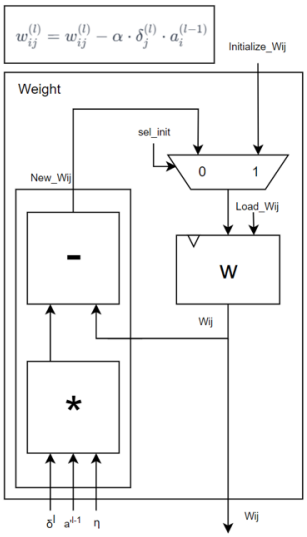

Figure 5. Neuron Weight Block

Starting off, the weight block had to be designed with back propagation in mind, since the weights need to be updated after each full pass through the network. It updates based on the sensitivity, activation function from the next layer, and networks learning rate. The values mentioned previously will be multiplied and subtracted from the previous weight in order to calculate and use the new adjusted weight. With the updated weight calculated, it

is fed into a mux, where that updated value will be output from the weight block component until it gets updated again.

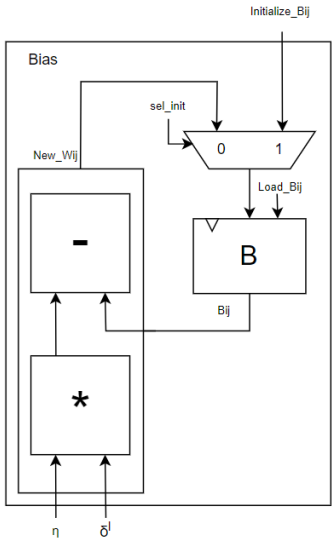

Figure 6. Neuron Bias Block

Despite the structures of the bias block and the weight block being similar in structure, their purpose in the network is very different. The bias block is meant to allow the network to adjust the weighted sum for a neuron. With the adjustment, the network is able to offset and shift the activation functions based on the sensitivity and the overall network's learning rate.

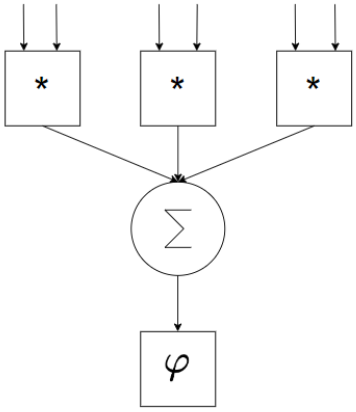

Figure 7. Neuron Weighted Summer

Moving on to the neurons weighted summer, which includes a few different functions. It first multiplies each given input with a weight that is coming from a certain weight block. After each input and weight are multiplied, they are summed together and passed through the ReLU activation function. which gives the resulting output of this particular neuron.

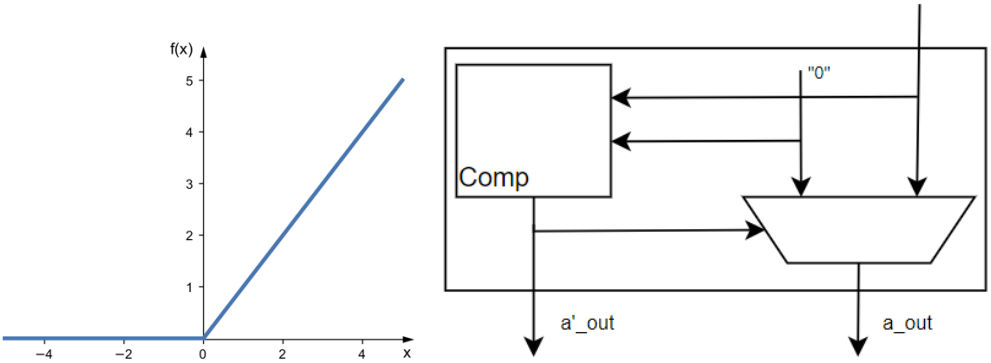

Figure 8 and Figure 9. Activation Function and Block (ReLU)

As stated previously, the result of the weighted summer feeds into the activation block. The ReLU activation function is piecewise linear, with a linear output for $x \geq 0$ and zero for $x < 0$. This block outputs not only the value of the activation, but the derivative of it as well. By using a ReLU instead of another nonlinear activation function, computing the derivative of a linear function results in a value. Figure 9 is the hardware implemented version of this function.

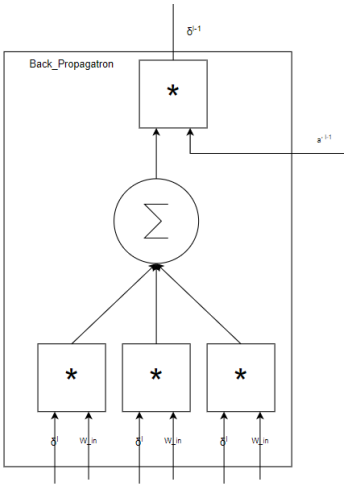

Figure 10. Backwards Propagation Block

The final part of a neuron is the backwards propagation block. This block's main purpose is to compute the loss based on the weights and biases of neurons from the next layer and sum them up. With that, the derivative of the activation function at the current forward pass neuron is multiplied with the summed result, which is needed for the weight block to update the weights according to the loss.

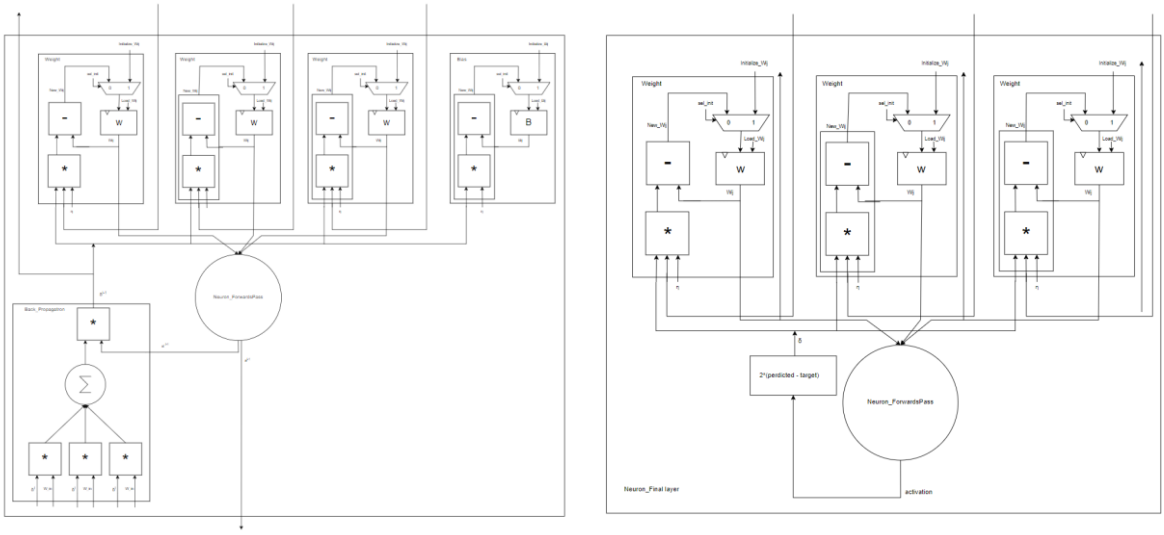

Figure 11 and Figure 12. Hidden Layer Neuron and Output Layer Neuron

Now that the main components of a neuron have been created, making different types of them based on their position in the network is essential. In neural networks, the number of weights going to each neuron depends on how many neurons were in the layer before it. Based on the hypothetical model from Figure 4, there are six neurons in the hidden layer and three neurons in the output layer. Since the neurons require a different number of components based on their location, different neurons were created based on their location. An example of how they would be different would be that neurons in the hidden layer need eight weights, which are passed from the input layer. This would require eight individual weight blocks per neuron. Neurons in the output layer get six weights sent to them from the hidden layer, meaning they only need six weight blocks.

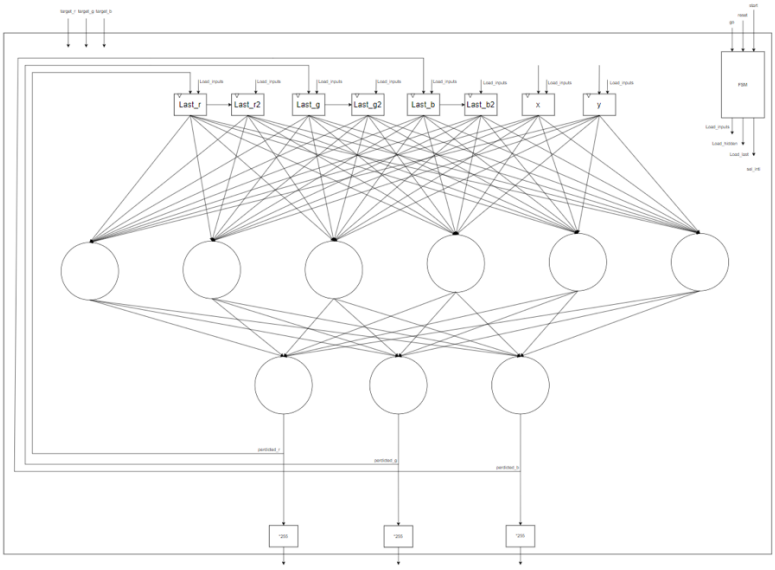

Figure 13. Neural Network Diagram

Now that each component in the network has been created, connecting them together was the next step. To reduce the amount of port mapping that needed to be done, generated loops were utilized for each neuron. Some decisions that were important on how effectively the network functioned were the width of the buses and utilizing registers to hold the proper sensitivities during backpropagation. For the bus width, most of the values, such as the learning rate, activation function output, and weights, were normalized. This means that the values' ranges were decimal values between zero and one. Fixed point numbers were utilized in order to properly interpret these small

decimal values. The reason the registers were so critical is because during the backpropagation process, each layer depends on the next calculated loss, which in turn modifies the weights for the next forward pass of the network. If the sensitivities were changed as it was adjusting the weights, it would make the backpropagation process useless. Having the sensitivities stay the same until the backpropagation process is complete allows for it to truly adapt and get closer to the intended RGB output.

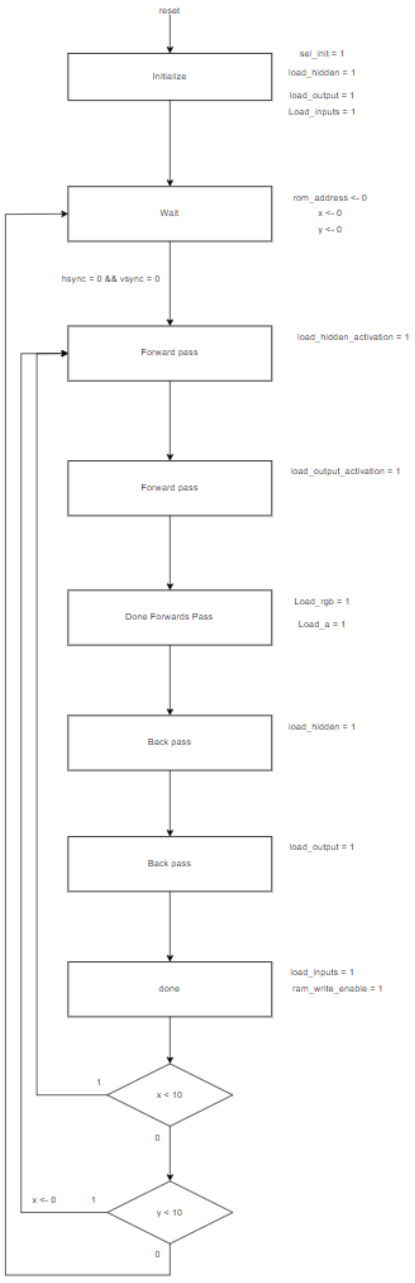

Figure 14. Network State Machine

The network state machine controls the order of operations of the neural network. The network starts in an initialization state, where it loads the input data into the network for computation. A state then will wait until both the horizontal and vertical syncs are equal to zero, while also setting the RAM address that is being written into to zero. In short, the VGA component is synced to the top left x and y coordinates of the screen, ensuring edits are made to the top left pixel. After this is detected, the first forward pass will occur in the hidden layer then another forward pass in the output layer. The new red, green, and blue color values will be stored in a register before being

loaded into the RAM. Back propagation will then take place to give new values to the weights for the next forward pass. This will occur in both the hidden layer and the output layer. The RGB values will now be loaded into the RAM and new inputs will be loaded into the network. The state machine calculates the next pixel's coordinates before doing the calculations over again. The process repeats for each pixel in the 10x10 image.

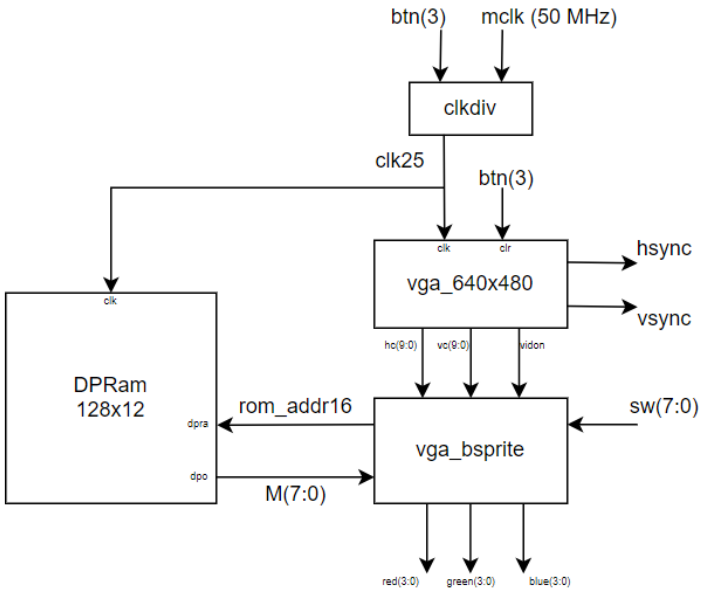

Figure 15. VGA Component

The VGA component utilized was built off the sprite example demonstrated in class. It is capable of displaying an image in 8-bit color through VGA to a screen. It uses switches and buttons to control the location and resetting of the image being displayed. Components such as the "vga_bsprite" needed to be converted to support 12-bit color instead of the typical 8-bit color. One may notice the use of a dual port RAM instead of the typical image ROM shown during lecture. By utilizing a dual port RAM, the group was able to avoid unnecessary communications between the neural network itself and the VGA component. In this way, the network can read and write data to the RAM using the typical "a", "d", "we", and "spo" ports. The VGA component would simultaneously be able to read data from the RAM using the "dpra" and "dpo" ports. In practice, this means that the neural network would be able to overwrite old data onto the RAM as the VGA component displays it. As long as the network's write address was ahead of the VGA component's read address, it would be possible to constantly display the results from the network as it learns in real time. The use of a dual port RAM allows for a much simpler and smarter way to accomplish the feature of watching a neural network learn.

4. Results

After building the control unit for the neural network, we implemented it with the datapath in an attempt to finish the project. After lengthy sessions of debugging and looking over each individual component, it was concluded we could not fully complete the neural network. Each component up to the overall structure of the network were simulated and compared to hand calculated values to ensure the components were correctly outputting values.

However, when it was time to put everything together, there simply was not enough time to fully fix the remaining errors in our project. The following figures display the testbenches built for the component of our network.

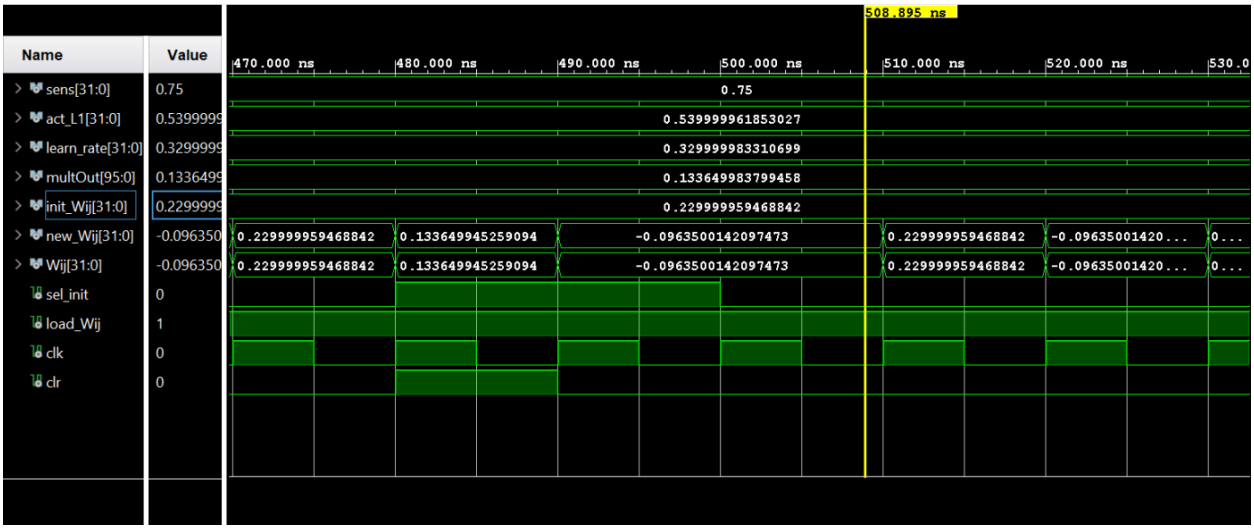

Figure 16. Neuron Weight Block Simulation

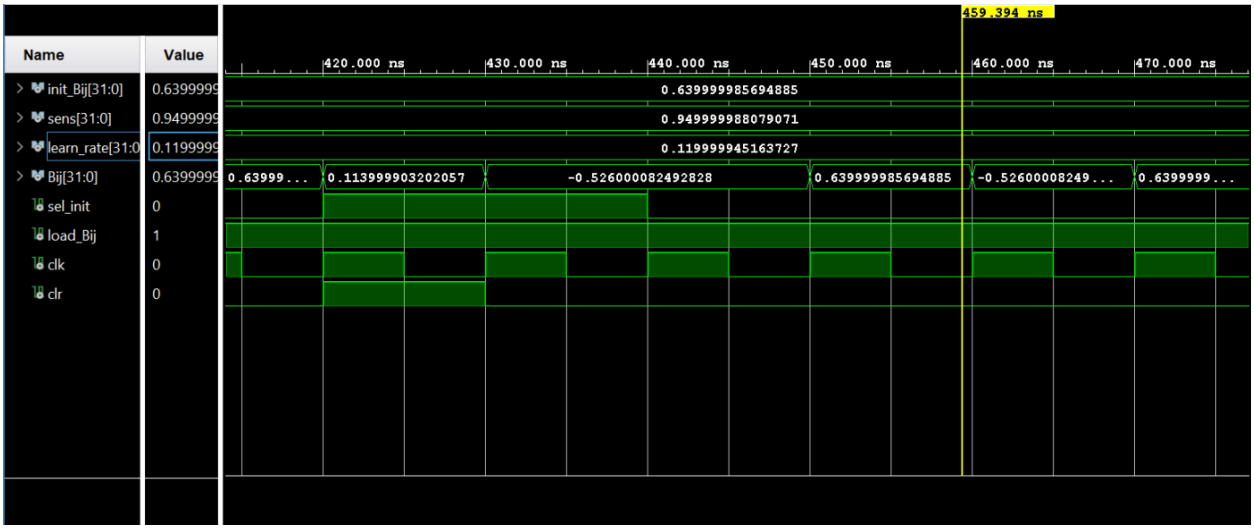

Figure 17. Neuron Bias Block Simulation

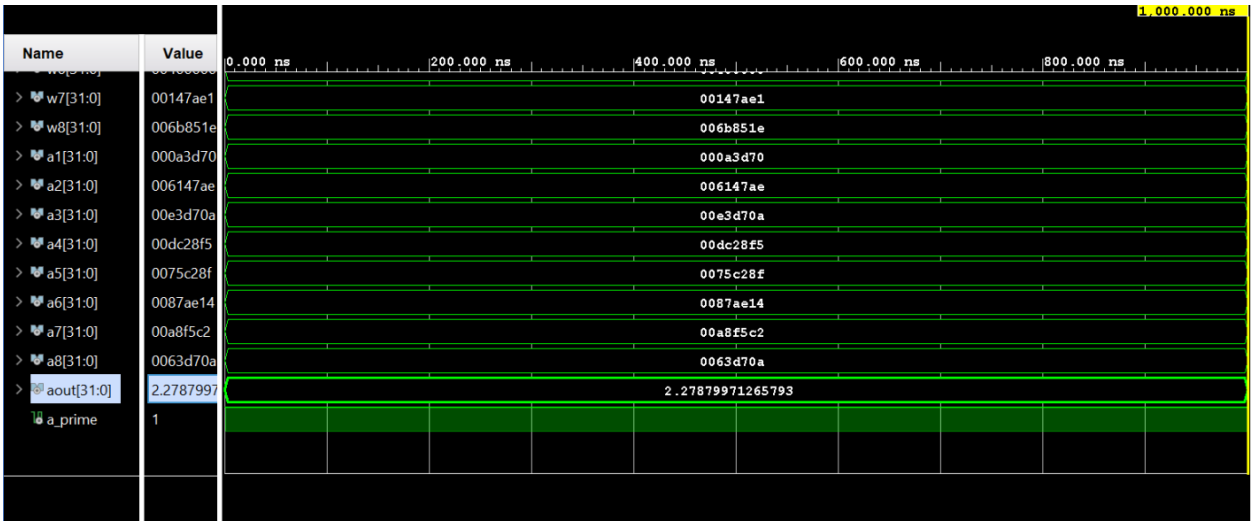

Figure 18. Neuron Weighted Summer Block Simulation

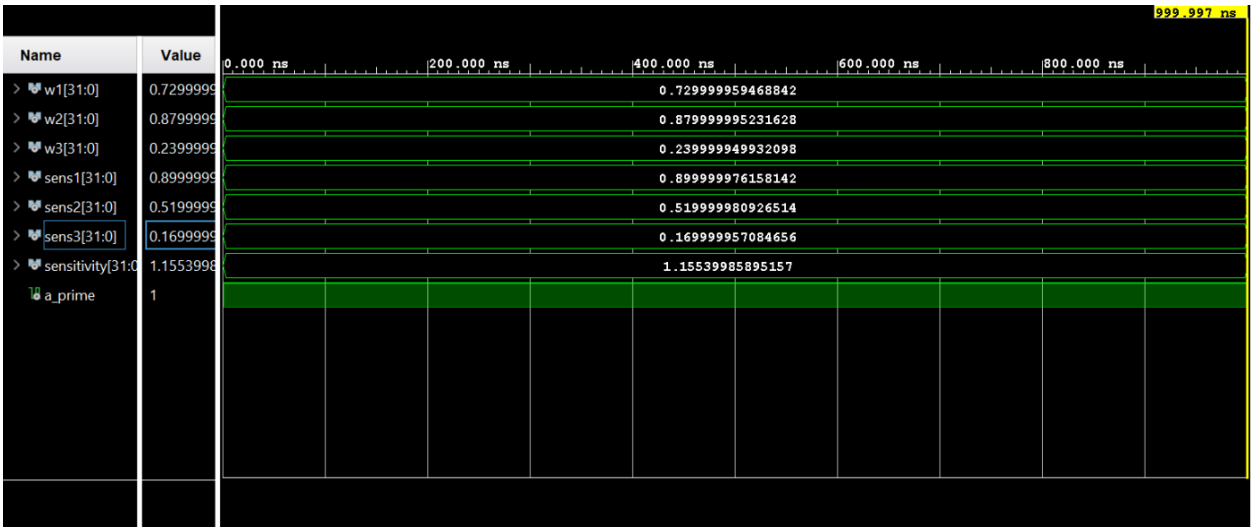

Figure 19. Back Propagation Neuron Block Simulation

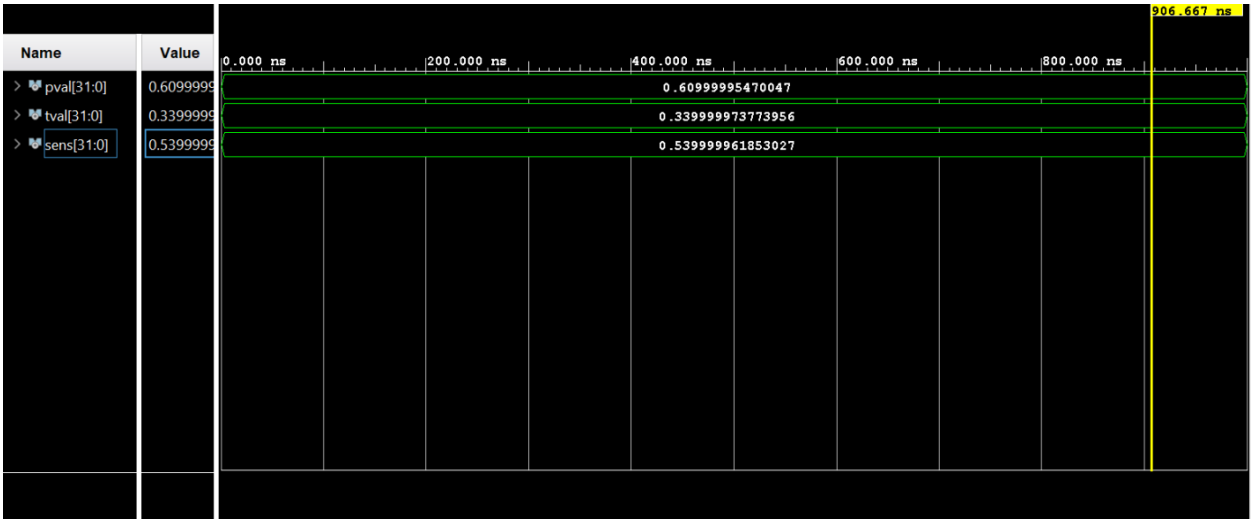

Figure 20. Final Layer Sensitivity Block Simulation

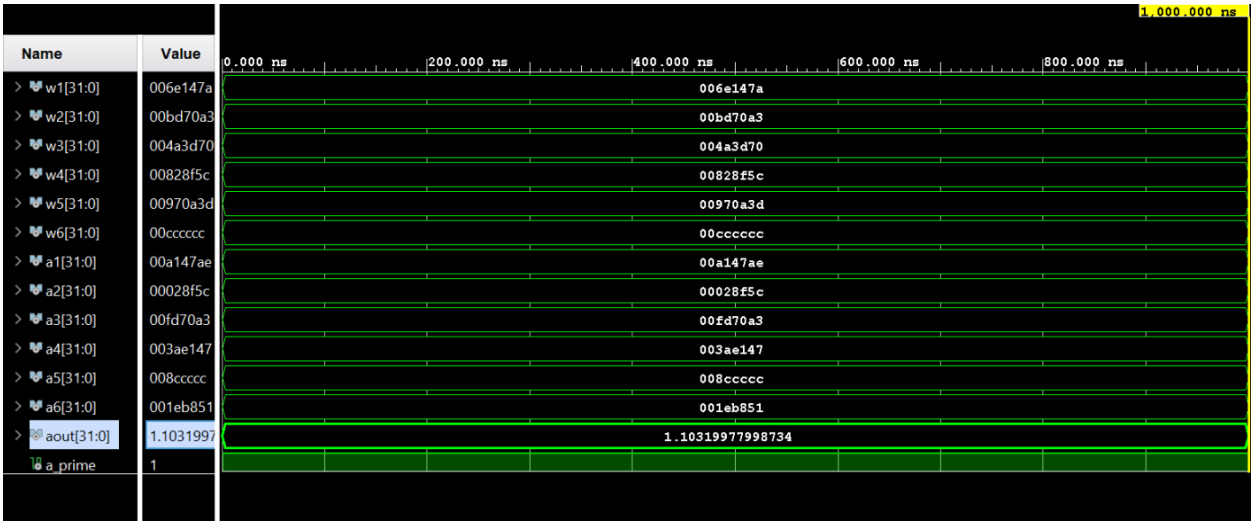

Figure 21. Output Layer Weighted Summer Neuron Simulation

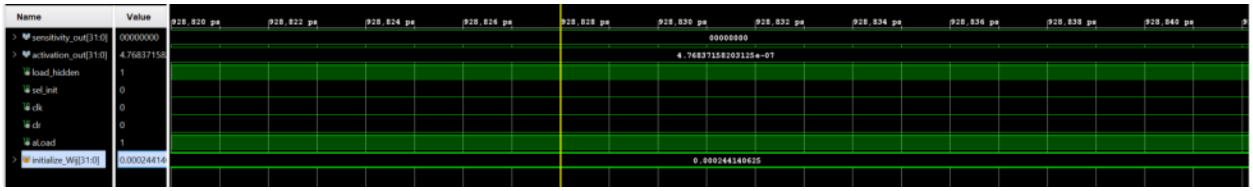

Figure 22. Hidden Layer Neuron Simulation

Each component was simulated through manually applying inputs. Since most inputs were decimal numbers between zero and one, it was necessary to find out how to represent decimal numbers in binary. To do this, we tried using floating point math, but the outputs still did not match our hand calculations. We realized

multiplication of different sized vectors were going to change where the decimal point was located, and therefore, floating point was not the answer. We then tried fixed point values, where the first eight bits of the thirty-two-bit vectors were the integer values, set to zero, and the remaining twenty-four were the decimal representation. With this, every output remained in the same format, and we could still have enough decimal places to have precise output values. In doing so, our simulated outputs equated the hand calculated values we got from using the same input values in our used formulas.

A few more problems encountered in the process of completing our project included how we wanted to structure our network and how to use generated loops. Previously, we discussed each of these topics. In order to get there, we made appointments with Professor Hanna and reached out to our TAs to discuss potential ways to complete each.

If given extra time, we would further debug our topmost file to get the network fully working. This could be done with one extra day to sit down and look deep into our files and the error messages given. Additionally, an expansion of this project would include blowing up our image from 10x10 to 100x100 to allow our image and renderings to appear more detailed and viewable.

5. Conclusions

In conclusion, this project aims to harness the power of neural networks for real-time image processing, specifically targeting a 10x10 pixel image display on a VGA screen using the Nexys A-7 FPGA development board. The introduction provides a foundational understanding of neural networks and their potential applications, highlighting the significance of efficient image processing in various fields.

As we delve into the subsequent sections of this report, we will explore the intricate details of the project, from its architectural planning to the VHDL-based programming of the neural network. The challenges posed by the constraints of embedded systems and the solutions devised to address them will be thoroughly examined, providing insights into the intelligent design practices implemented to optimize performance and resource utilization on the limited FPGA platform.

This project not only serves as a technical endeavor but also as a practical demonstration of the Nexys A-7's capabilities in the realm of real-time image processing applications. The report, structured to cover background, planning, design, challenges, and results, aims to provide a comprehensive overview of the entire undertaking. Through this work, we aspire to contribute to the advancement of FPGA-accelerated computation, particularly in the context of image-based applications relevant to fields like robotics and surveillance.

6. References

(1) *Digital Design Using Digilent FPGA Boards - VHDL/Vivado Edition*, by Richard E. Haskell and Darrin M. Hanna, LBE Books, 2018.

(2) *Advanced VHDL Graphics Using Digilent FPGA Boards*, by Richard E. Haskell and Darrin M. Hanna, LBE Books, 2014.