

Copyright
by
Isaac Westby
2020

FPGA ACCELERATION ON MULTILAYER PERCEPTRON (MLP) NEURAL
NETWORK FOR HANDWRITTEN DIGIT RECOGNITION

by

Isaac Westby, BS

THESIS

Presented to the Faculty of
The University of Houston-Clear Lake

In Partial Fulfillment

Of the Requirements

For the Degree

MASTER OF SCIENCE

in Computer Engineering

THE UNIVERSITY OF HOUSTON-CLEAR LAKE

MAY, 2020

FPGA ACCELERATION ON MULTILAYER PERCEPTRON (MLP) NEURAL
NETWORK FOR HANDWRITTEN DIGIT RECOGNITION

by

Isaac Westby

APPROVED BY

Xiaokun Yang, PhD, Chair

Hakduran Koc, PhD, Committee Member

Jiang Lu, PhD, Committee Member

RECEIVED/APPROVED BY THE COLLEGE OF SCIENCE AND ENGINEERING:

David Garrison, PhD, Associate Dean

Miguel A Gonzalez, PhD, Dean

Dedication

I would like to dedicate this thesis to my Dad, thank you for all your love and support through the years.

Acknowledgements

I would like to thank Dr. Xiaokun Yang for all his help on this project. I would like to thank him for being so helpful and understanding through the entire time. Without his support I would not have been able to accomplish all that I have in this project.

ABSTRACT

FPGA ACCELERATION ON MULTILAYER PERCEPTRON (MLP) NEURAL NETWORK FOR HANDWRITTEN DIGIT RECOGNITION

Isaac Westby
University of Houston-Clear Lake, 2020

Thesis Chair: Xiaokun Yang, PhD

This dissertation presents a hardware implementation of a Multi-Layer Perceptron (MLP) network used for the purpose of low-latency, high-accuracy digit recognition. The accuracy of various network designs was compared in Python, and the final network design was comprised of 784 input neurons, a single hidden-layer of 12 neurons, and an output layer of 10 neurons. The weights and biases of this network were then trained using the Modified National Institute of Standards and Technology (MNIST) handwritten digit data in Python using the stochastic gradient descent method. This network design was then tested in software for the digit recognition accuracy of half (16-bit), single (32-bit), and double (64-bit) precision inputs. These all gave nearly the same results of (93.26, 93.25, and 93.25%) digit recognition accuracy respectively. This design was then implemented in hardware using the Verilog Hardware Description Language (HDL). This novel design uses a custom timing structure along with single-precision, floating-point

IPs from Vivado for multiplication, addition, subtraction, accumulation, exponential, and reciprocal. Results show a speedup of 40.3967 over the fastest software execution, and 127.219 over the slowest software execution. The results of the synthesis were found for the Kintex-Ultrascale FPGA, part xcku035-sfva784-1LV-I. These results showed a utilization of 44,668 Look Up Tables (LUT), 14,274 Flip Flops (FF), and 604 Digital Signal Processors (DSP), for utilization of 21.99%, 3.51%, and 35.53% respectively. Compared with related works, our proposed work provides the lowest latency for digit recognition, with a speedup of 61 and 42 over these works. Further compared to these related works, our design is between the two in accuracy and resource utilization, showing a tradeoff between design complexity and digit recognition accuracy. Conclusions of our research are that our proposed design presents a high-accuracy, low-latency digit recognition network. Our proposed design allows for further customization to fit with a future user's needs.

TABLE OF CONTENTS

List of Tables	x
List of Figures	xi
CHAPTER 1: INTRODUCTION	1
1.1 Background	1
1.2 Related Works	2
1.3 Design Process	4
1.4 Structure of Dissertation	4
CHAPTER 2: PROPOSED DESIGN OF THE NETWORK	6
2.1 Neural Networks	6
2.2 Sigmoid Neurons	7
2.3 Training the Neural Network	7
2.4 Running the Python Code	10
2.4.1 Resources Needed	10
2.4.2 Steps to Run the Code	10
2.4.3 Exporting Weights and Biases to CSV	13
2.5 Finding the Best Design	14
2.5.1 Comparing Different Networks	14
2.5.2 Adjusting Epoch, mini_batch_size, and Learning Rate	18
2.6 Final Network Design	22
CHAPTER 3: SOFTWARE IMPLEMENTATION	24
3.1 Matlab Implementation	24
3.2 Matlab Results	28
3.3 Accuracy Comparison	32
CHAPTER 4: HARDWARE IMPLEMENTATION	34
4.1 Hardware Design Architecture	34
4.2 Final Hardware Architecture	39
4.3 Timing and RTL Design	40
4.3.1 How the Counters are Used	40
4.3.2 RTL Design	44
4.4 Vivado IPs	46
4.4.1 Instantiating the IPs	46
4.4.2 Distributed Memory Generator IP	49
4.4.3 How to Create COE files Using Matlab	50
4.5 Running the Project	51

4.5.1 Creating the Project Using TCL	52
4.5.2 Running the TCL Script.....	52
4.5.3 Synthesis in Vivado	53
4.5.4 Generating the Waveform in Vivado	54
4.5.5 Interpreting Results in Vivado	55
CHAPTER 5: RESULTS	63
5.1 Execution Time on FPGA.....	63
5.2 Resource Cost on FPGA	64
5.3 Power Cost on FPGA.....	67
5.4 Comparison to Related Works	68
5.4.1 Accuracy Comparison.....	68
5.4.2 Speed Comparison	68
5.4.3 Utilization Comparison	69
5.4.4 Summary of Comparison	70
CHAPTER 6: CONCLUSIONS AND FUTURE WORK.....	71
6.1 Conclusions	71
6.2 Future Work	72
REFERENCES	74

LIST OF TABLES

Table 2.1 The data corresponding to graph in figure 2.4.....	16
Table 2.2 The data corresponding to graph in figure 2.5.....	17
Table 3.1 A layout of the weights of the first hidden layer.	24
Table 3.2 A layout of the input pixels of the first hidden layer.	25
Table 3.3 A layout of the biases of the first hidden layer.	25
Table 3.4 An example layout of the weights1.csv file.....	26
Table 3.5 An example layout output neuron biases.....	26
Table 3.6 Corresponding data used for comparison of ‘strength’ of single and double precision datatypes.....	30
Table 3.7 Corresponding data used for comparison of ‘strength’ greater than 0.9 of single and double precision datatypes.	31
Table 4.1 A comparison of the resources needed for a pipelined vs non-pipelined design.	36
Table 4.2 A comparison of the resources needed for a pipelined, non-pipelined design, and pipelined with 98 multipliers designs.....	38
Table 4.3 A table showing the results of the network with an input of a handwritten 7.	58
Table 4.4 A table showing the results of the network with an input of a handwritten 2.	60
Table 4.5 A table showing the results of the network with an input of a handwritten 1.	62
Table 5.1 A comparison of the execution times in software and hardware.....	64
Table 5.2 The utilization results of the hardware synthesis.....	65
Table 5.3 A further breakdown of the utilization results.	66
Table 5.4 Accuracy comparison to related works.....	68
Table 5.5 Speed comparison to related works.	69
Table 5.6 A utilization comparison to related works.....	70

LIST OF FIGURES

Figure 2.1 A visualization of the Gradient descent curve.....	9
Figure 2.2 The output that you should see when running the Python code.	12
Figure 2.3 An image showing the commands to save the weights and biases to a CSV file.	13
Figure 2.4 A graph comparing accuracy of different networks with 1 hidden-layer of various numbers of neurons.....	15
Figure 2.5 A graph comparing accuracy of different networks with 2 hidden-layers of various numbers of neurons.	17
Figure 2.6 A graph comparing networks with a single hidden layer and two hidden layers.	18
Figure 2.7 A graph comparing the digit recognition accuracy with the mini-batch size.	19
Figure 2.8 A graph comparing the digit recognition accuracy with the learning rate.....	20
Figure 2.9 A graph comparing the digit recognition accuracy with varying parameters.	21
Figure 2.10 A figure showing the final network design.	23
Figure 3.1 An example output when running the program.	27
Figure 3.2 An example output when running the program with an input of 3.	28
Figure 3.3 Comparison of half, single, and double precision results in Matlab.	29
Figure 3.4 Comparison of the ‘strength’ of single and double precision data-types.	30
Figure 3.5 Further comparison of the ‘strength’ greater than 0.9 of single and double precision data-types.....	31
Figure 4.1 A visual representation of the multipliers and adders going into the first layer hidden neurons.	34
Figure 4.2 A visual representation of the results from the multipliers and adders going into the final operations of the neuron.	35
Figure 4.3 A visual representation of the operations of a single output layer neuron.	36
Figure 4.4 A visual representation of the first stage multiplication with 98 multipliers.	37
Figure 4.5 A visual representation of the accumulation and final operations used after the 98 multipliers.	38

Figure 4.6 A visual representation of all the components of the design put together.	39
Figure 4.7 A visual diagram of the timing of operations aligned with counters.	43
Figure 4.8 A simplified hardware translation of the enable design.	45
Figure 4.9 An example COE file for the hidden layer biases.	50
Figure 4.10 The line in the .tcl file that needs to be changed to correspond to your directory.	52
Figure 4.11 An example of the direct path that needs to be changed.	53
Figure 4.12 Vivado waveform with all counters and their corresponding enable signals.	55
Figure 4.13 Vivado waveform showing when cnt6 starts counting.....	56
Figure 4.14 Vivado waveform showing the final results of the network.....	57
Figure 4.15 Vivado waveform showing the final results of the network for input of 2.	59
Figure 4.16 Vivado waveform showing the final results of the network for input of 1.	61
Figure 5.1 A graph of the utilization results.	65
Figure 5.2 An image of the power breakdown.	67
Figure 5.3 A further breakdown of the power consumption.....	67

CHAPTER 1: INTRODUCTION

1.1 Background

The subject of creating a computational model for neural networks (NNs) can be traced back to 1943 [45, 46]. Until 1970, the general method for automatic differentiation of discrete connected networks of nested differentiable functions has been published by Seppo Linnainmaa [47,48]. In 1980s, the VLSI technology enabled the development of practical artificial neural networks. A landmark publication in this field is a book authored by Carver A. Mead and Mohammed Ismail, titled “Analog VLSI Implementation of Neural Systems” in 1989 [49].

The real-world applications of NNs mainly appeared after 2000 [27]. In 2009, the design on the network with long short-term memory (LSTM) won three competitions of handwriting recognition without any prior knowledge about the three languages to be learned [52,53]. And in 2012, Ng and Dean created a network being able to recognize higher-level concepts such as cats [50].

Today, NNs have shown great ability to process emerging applications such as speech/music recognition [1,2], language recognition [3,33], image classification [4,5], video segmentation [6,19], and robotic [41, 43]. With the artificial intelligence (AI) chip market report published in May 2019, the global AI chip market size was valued at \$6,638.0 million in 2018, and is projected to reach \$91,185.1 million by 2025, growing at Compound Annual Growth Rate (CAGR) of 45.2% from 2019 to 2025 [54]. Therefore, to make NNs high speed and efficient will have a profound impact on transportation, sustainability, manufacturing, city services, banking, healthcare, education, entertainment, gaming, defense, criminal investigation, and many more.

Prior researches on NNs were mainly based on the software implementations, which can be very effective but require intensive CPU operations and memory bandwidth to achieve the desired results. The increasing computing power including but not limited to application-specific integrated circuit (ASIC), field-programmable gate array (FPGAs), and system-on-chips (SoCs) has been boosting the use of larger networks in image and visual recognition [23, 51]. These innovative technologies will contribute to making AI a reality, particularly to the time- and resource-constrained applications such as smart cars, industrial control systems, fraud detection with financial service, Internet of Medical Things (IoMT), speech recognition, natural language processing (NLP), automated speech recognition, and unmanned surveillance vehicles.

In addition, Internet of Things (IoT) devices have taken in data, processed it, and sent it off to the cloud where the results have been calculated, then these results are sent back to the IoT device [28,29,30]. This has been successful because the IoT device is able to remain simple, needing only to process and send off results. As the number of IoT device increases, and drives pressure on finite bandwidths, edge computing has emerged as a new trend [24, 40]. Edge computing allows for processing of data at the network edge rather than sending the data off and processing remotely. Software solutions are good up to a point, but hardware acceleration can provide a faster, lower power solution [31,32]. Under this context, in this paper a case study of the hardware acceleration on digit recognition with a FPGA is presented.

1.2 Related Works

To date, many designs on digit recognition have been presented on the algorithm level [11,7] and hardware level [9,10,19]. For example, a recurrent neural network (RNN) has been proposed in [21] to recognize digits, and in [20] the designs on Deep

Neural Network (DNN), Convolutional neural networks) (CNN), and Bidirectional Recurrent Neural network (RNN) have been implemented and evaluated. As a result, the accuracy can reach 99.6% with the CNN and 97.8% with the four-layer DNN. The accuracy of RNN is 99.2%. All these researches were focused on exploring the accuracy of the implementations on algorithm and software level.

On hardware level, many prior researches focused on the design with high speed [22,8], low-energy cost [38,39,44], and security [36, 37]. In this thesis we concentrating on finding the optimal acceleration of digit recognition with FPGA corresponding to the quality edge. The latest implementation on multilayer perceptron (MLP) in [12] has shown the computation latency as 3.8 seconds including both training and inference. Four implementations were provided in this work, including four-bit, five-bit, six-bit, and eight-bit designs with FPGA. The eight-bit MLP consumed 34k logic elements but achieved the highest accuracy (89%) compared with the other implementations; and the four-bit design can reduce the slice count to 20k with 9% accuracy decrease. Different from the fully connected network as MLP, CNN is created with layers sparsely connected or partially connected. In [13], the digit recognition on an eight-bit CNN has been presented. Experimental results showed that the proposed hardware achieved a latency of 9.4 seconds on the design with sequential channel and 2.2 seconds with the parallel channel. The hardware costs are 20k and 98k respectively by using the sequential channel and parallel channel on FPGA. Both of the designs can achieve an accuracy over 90%. Compared with [12] and [13], a single-hidden-layer MLP is proposed to continually reduce the complexity of the design on digit recognition with FPGA.

1.3 Design Process

The goal of this project was to design a neural network that could recognize handwritten digits with an accuracy of $>90\%$ and with a latency of $< 3.5\text{ms}$ for each recognition. The implementation is based on the database of Modified National Institute of Standards and Technology (MNIST), which was developed by Yann LeCun, Corinna Cortes and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem [15,14]. The network created was trained and tested based upon this MNIST data. The entire design process took place in a linear fashion and was composed of three major parts.

1. Decide on the network design to be used.
2. Test the network design in software using Matlab.
3. Build and test hardware implementation of the design in Vivado.

These are obviously simplified steps in the design process, but they reflect the major portions of the project. Each of these parts were internally iterative, (i.e. for step 1 various network designs were considered before a final one was chosen), but each step needed to be completed before moving on to the next.

1.4 Structure of Dissertation

The remaining structure of the dissertation follows the parts of the design process through the completion of the project. In Chapter 2 I will be describing the proposed design of the network. In this chapter I will converge on a network design and describe how I reached this decision. In Chapter 3, I will be taking the proposed design described in Chapter 2, and implementing it in Matlab. This chapter will then compare the digit recognition accuracy of the network in Matlab using various floating-point precision inputs. Chapter 4 describes the hardware implementation and everything that went into

taking the design that was used in Matlab and converting it to RTL. Chapter 5 will then take the results of the hardware implementation and compare them to the results of the software implementation as well as to related works. Finally, in Chapter 6 the final conclusions of the project and future work are described.

CHAPTER 2:

PROPOSED DESIGN OF THE NETWORK

This chapter is important in showing the process of choosing network design that we did. We first cover some of the background on Neural Networks, then move onto discussing how the network is trained. Finally, Python code from [26] is used to setup and train the network is used to compare different networks with different training parameters.

2.1 Neural Networks

The history of Neural Networks can be traced back to 1943 when McColluch and Pitts wrote a paper on how neurons work. In order to more easily explain this complex phenomenon, they used electrical circuits to show it [25]. It was with this previous work in mind, that the perceptron was created by Frank Rosenblatt. The perceptron takes in multiple binary inputs, and then outputs a single binary output. These inputs are then multiplied by their corresponding weights, and summed together. That sum is then compared to a bias value in order to determine the output of the neuron. The equation for the output of perceptron is shown.

$$output = \begin{cases} 0 & \text{if } \sum_i w_i * x_i + b \leq 0 \\ 1 & \text{if } \sum_i w_i * x_i + b > 0 \end{cases}$$

where x is the input, w is the weight, and b is the bias.

This network design described by Rosenblatt gave us the ability to tweak or ‘learn’ new weights and biases in order to get the desired output. It is this idea that is the basis for most of the artificial neural networks that we see today.

2.2 Sigmoid Neurons

The efficacy of the perceptron starts to break down when you are training it. Because the output is binary, any small change in the input weights and biases can affect the output. Due to the nature of the output being binary, it makes it very hard to ‘train’ these perceptron networks. This is where the sigmoid neuron comes in. The sigmoid neural network works similar to the perceptron network, except that the output is a value between 0 and 1.0, instead of a binary value of 0 or 1. The output for the sigmoid neuron is shown below.

$$\text{sigmoid neuron output} = \frac{1}{1 + \exp(-(\sum_i w_i * x_i) + b)}$$

where w are the weights corresponding to the inputs x , and b represents the bias.

This sigmoid neuron solves the problem with training. By making small changes to the weights and biases of the sigmoid neuron, you are able to make small changes to the output, eventually converging on a ‘correct’ or most effective set of weights and biases.

Because of this fact, sigmoid neurons are used in the network that I am using. In order to simulate this network, I will need to run the sigmoid function for all neurons in the hidden and output layers. Also, to clarify a point of confusion here, as stated in [26], networks made up of sigmoid neurons are often referred to as Multi Layer Perceptron (MLP) networks. Any future reference to an MLP network is referring to a network made of these sigmoid neurons.

2.3 Training the Neural Network

As stated above, because we are using the sigmoid neuron in our network, we have the ability to train the network by making small changes to the weights and biases. In order to train our network, we are using the MNIST handwritten digit dataset. This

dataset has 60,000 handwritten digits with corresponding labels that can be used to train the network. There is then a separate set of 10,000 different handwritten digits with labels that can be used to test the network. The test data set also comes from a completely different group of people, so a person who drew a '2' for instance in the training set, will not have drawn any digits in the testing set.

The methods and ideas used in training the neural network are taken from the Neural Networks and Deep Learning Book by Michael Nielson [26]. In training the network, we have a cost function dependent on the weights and biases shown below.

$$C(w, b) \stackrel{\text{def}}{=} \frac{1}{2n} \sum_x ||y(x) - a||^2$$

where $y(x)$ is the results array of 10 values where the correct result has one value equal to 1 and the rest equal to 0. For example $y(1) = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0)^T$. The value of a represents the output vector of the network. The value of n is the total number of training inputs, and x is the value of the training input.

The idea here is that we want to minimize this cost function. This would mean that our results are as close as possible to $y(x)$, which is the actual value. With this cost function, we are able to provide a result that is dependent on the two inputs (weights and biases) which are to be changed in training. The way that the Cost function is minimized is through the process of gradient descent. In this process, you are taking the gradient of the cost function repeatedly, such that with each step you are trying to decrease the Cost function the most. If we look at the image of the cost function, with two direction vectors from [26], shown on the next page.

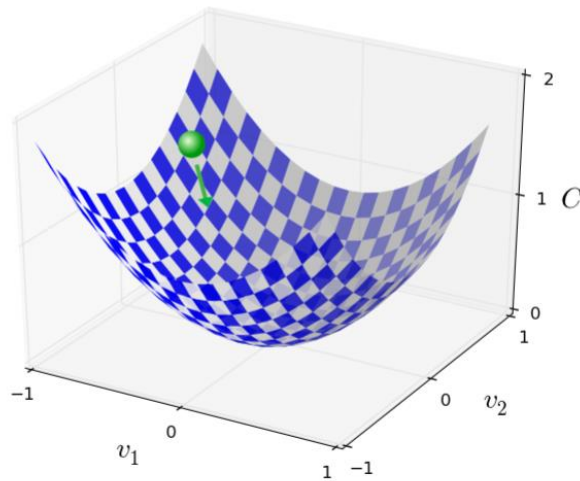


Figure 2.1
A visualization of the Gradient descent curve.

Looking at this image, the idea with gradient descent is to change the values of v_1 and v_2 , that the green ball can ‘roll’ down to the minimum value of C . Through this gradient descent, we are updating the values for all the weights and biases. In order to compute the gradient for the cost function you need to compute the gradient of each individual function individually. This is a weak point of gradient descent when you have a large sample size used for training.

Stochastic Gradient descent is used as a way to speed up training. In this method we are doing the training based on a small random sample size, then average the entire thing together. This is different than calculating the entire gradient for all inputs in gradient descent method. In this training, we are choosing a ‘mini-batch size’ which is user defined and made up of randomly chosen inputs. The way that the training works is it choose a mini-batch of inputs and trains with those, once those have all been used, it then picks another mini-batch of inputs until all the inputs have been exhausted. Once all the inputs have been exhausted, this is called an ‘Epoch’. Once an Epoch has been completed, the process starts all over again in the next Epoch. The analogy that is used in

[26] is to think of this like political polling, where we are just looking at a small sample size that is representative of the whole more or less. This process is implemented in the Python code provided by [26] and used for training my network.

2.4 Running the Python Code

2.4.1 Resources Needed

In generating the weights and biases for the sigmoid Neural Network, Python was used. In my case, I have done this task using Python 2.7.17, which can be downloaded at <https://www.python.org/downloads/release/python-2716/> .

The code that is used to accomplish this task is written by [26] and can be found at <https://github.com/mnielsen/neural-networks-and-deep-learning> .

You can also clone the entire repository using the following command:

```
git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git
```

From this repository, the files that will be used are:

neural-networks-and-deep-learning/data/mnist.pkl.gz

neural-networks-and-deep-learning/src/mnist_loader.py

neural-networks-and-deep-learning/src/network.py

These files also use a Python Library called Numpy, which can be downloaded at <https://www.scipy.org/install.html> .

2.4.2 Steps to Run the Code

To run the code to generate the weights and biases of a network of your design, follow these steps.

1. Move the above three files (mnist.pk.gz, mnist_loader.py, and network.py) into your working Python directory. In my case, C:\Python27 . Make sure you have installed Numpy into this directory if you did not already have it installed.

2. Open 'IDLE (Python GUI)'. This should open up a Python shell.

3. Execute the following in the python shell:

```
import mnist_loader
```

This imports the helper file, mnist_loader.py that unpacks the MNIST data.

4. Execute the following in the python shell:

```
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
```

This again is setting up the MNIST data that is used for training, validation, and testing.

5. Execute the following in the python shell:

```
import network
```

This imports the network python file.

6. Execute the following in the python shell:

```
net = network.Network([784, 30, 10])
```

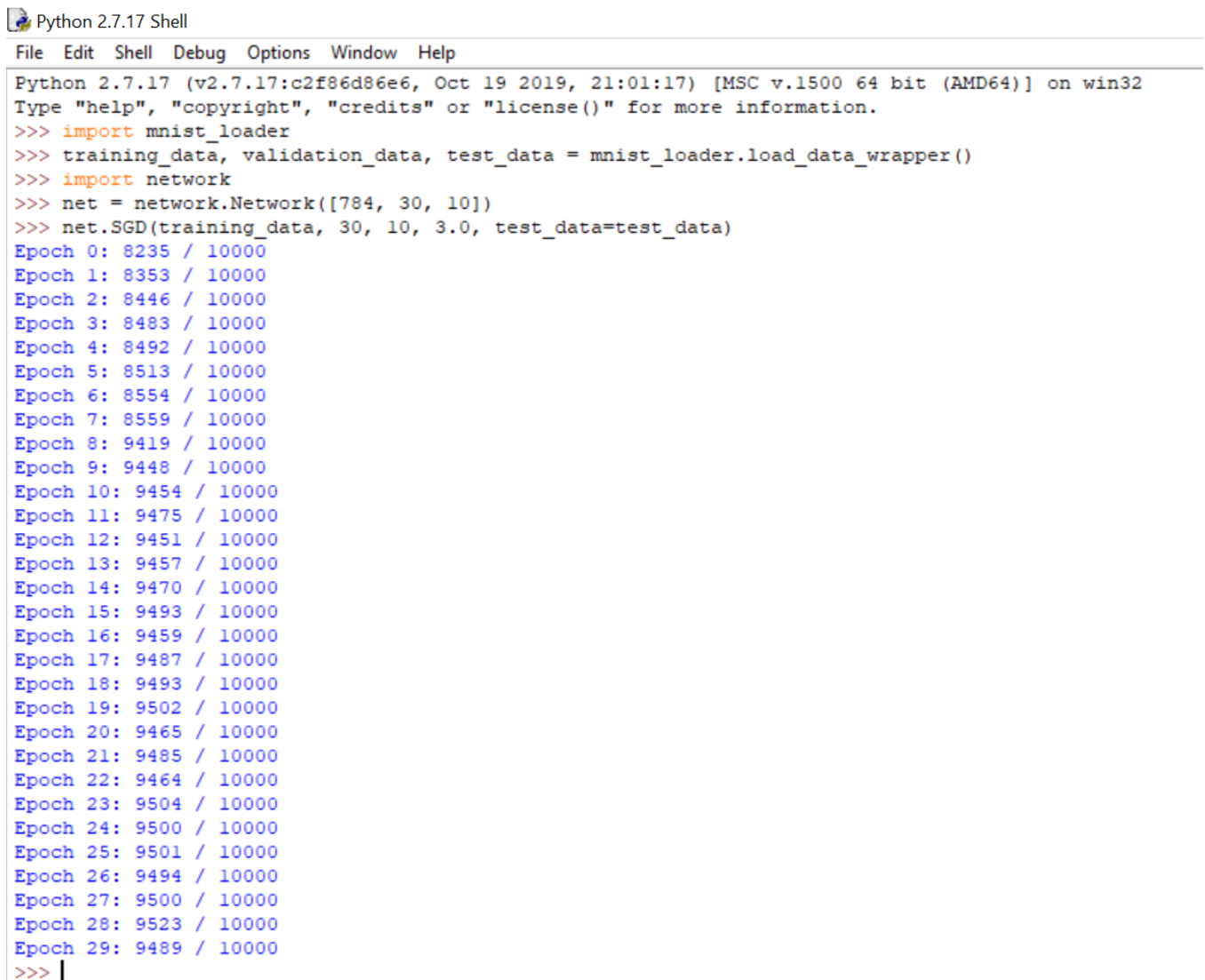
This command is setting up the Neural Network that we are using here. The number of input neurons is 28-pixel x 28-pixel = 784. The hidden-layer neuron value here is 30, and the output layer neuron value is 10. The input and output layer values need to remain static, but the number of neurons in the hidden layer can be changed, as well as the number of hidden layers (i.e. [784, 30, 30, 10]).

7. Execute the following in the python shell:

```
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

This command starts the training of the neural network by Stochastic Gradient Descent. The variables represent, the training data that is being used, the number of epochs over which it is trained (30), the mini-batch size used (10), the learning rate (3.0), and the test data being used.

Once the above steps have been completed, you should see an output similar to below:



```
Python 2.7.17 Shell
File Edit Shell Debug Options Window Help
Python 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019, 21:01:17) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
>>> import network
>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
Epoch 0: 8235 / 10000
Epoch 1: 8353 / 10000
Epoch 2: 8446 / 10000
Epoch 3: 8483 / 10000
Epoch 4: 8492 / 10000
Epoch 5: 8513 / 10000
Epoch 6: 8554 / 10000
Epoch 7: 8559 / 10000
Epoch 8: 9419 / 10000
Epoch 9: 9448 / 10000
Epoch 10: 9454 / 10000
Epoch 11: 9475 / 10000
Epoch 12: 9451 / 10000
Epoch 13: 9457 / 10000
Epoch 14: 9470 / 10000
Epoch 15: 9493 / 10000
Epoch 16: 9459 / 10000
Epoch 17: 9487 / 10000
Epoch 18: 9493 / 10000
Epoch 19: 9502 / 10000
Epoch 20: 9465 / 10000
Epoch 21: 9485 / 10000
Epoch 22: 9464 / 10000
Epoch 23: 9504 / 10000
Epoch 24: 9500 / 10000
Epoch 25: 9501 / 10000
Epoch 26: 9494 / 10000
Epoch 27: 9500 / 10000
Epoch 28: 9523 / 10000
Epoch 29: 9489 / 10000
>>> |
```

Figure 2.2
The output that you should see when running the Python code.

The values here show the number of correct digit classifications were made across each epoch with the weights and biases. In order to design the most accurate Neural Network, the values of the neurons, epochs, mini-batch size, and learning rate can be changed in order to find the weights and biases that are for that design.

2.4.3 Exporting Weights and Biases to CSV

Once you are content with the design of the network and the accuracy, you can export the values of the weights and biases by running the following commands:

```
import numpy

numpy.savetxt("weights0.csv", net.weights[0], delimiter=",")
numpy.savetxt("weights1.csv", net.weights[1], delimiter=",")
numpy.savetxt("biases0.csv", net.biases[0], delimiter=",")
numpy.savetxt("biases1.csv", net.biases[1], delimiter=",")
```

In the Python shell it should look like below:

```
>>> import numpy
>>> numpy.savetxt("weights0.csv", net.weights[0], delimiter=",")
>>> numpy.savetxt("weights1.csv", net.weights[1], delimiter=",")
>>> numpy.savetxt("biases0.csv", net.biases[0], delimiter=",")
>>> numpy.savetxt("biases1.csv", net.biases[1], delimiter=",")
```

Figure 2.3

An image showing the commands to save the weights and biases to a CSV file.

Make sure that the quotation marks used match as shown in the above image, I had a problem with this when copying/pasting the commands.

After executing these commands, you will have four separate CSV files with the weights and biases for your network.

2.5 Finding the Best Design

Once the method for finding the accuracy of the network with a trained set of weights and biases had been established, the goal was now to decide on a network design. In choosing the design, the goal was to have a high accuracy in digit detection, while maintaining as simple a design as possible.

In choosing the network design, there were a few things that would be forced to remain static due to the problem at hand. First, the input layer would require 784 neurons due to the fact that the MNIST digit images we would be using to train the network are $28\text{-pixel} \times 28\text{-pixel} = 784$ input pixels in size. The second thing that would need to remain static, is the fact there would be 10 output neurons. This value is static at 10 because there are 10 possible outputs (0-9) that would converge to a value of around 1 when that value is converged on by the network. The value of 10 outputs is chosen over a value of 4 outputs ($2^4=16$ possible outputs), because according to [26], the design of a system using 10 outputs rather than 4 produced empirically better results. This leaves the number of hidden layers, as well as the number of neurons in each hidden layer as the values that can be adjusted.

Once a network design (number of hidden layers and number of nodes in each layer) has been decided, the values of epochs used, `mini_batch_size`, and learning rate can be tweaked in order to find the best results. These values are static with values of `epoch = 30`, `mini_batch_size = 10`, and `learning rate = 3.0`, when comparing different network designs. Once a network design was chosen, these values were tweaked in order to find the most accurate set of weights and biases.

2.5.1 Comparing Different Networks

As stated above, various network designs were considered to find a network that would be able to be implemented with relative simplicity but still attain a high accuracy.

When testing different networks, the epoch, mini_batch size, and learning rate all remained the same, but the number of hidden layers and neurons in each layer were changed. The following 1-hidden layer networks were tested [784, 50, 10], [784, 30, 10], [784, 20, 10], [784, 16, 10], [784, 12, 10], [784, 10, 10], [784, 8, 10], [784, 5, 10], [784, 4, 10], [784, 3, 10], [784, 2, 10], [784, 1, 10]. Where the first number is the input layer number, the second number is the number of neurons in the hidden layer, and the third number is the output layer neurons. The results are shown below.

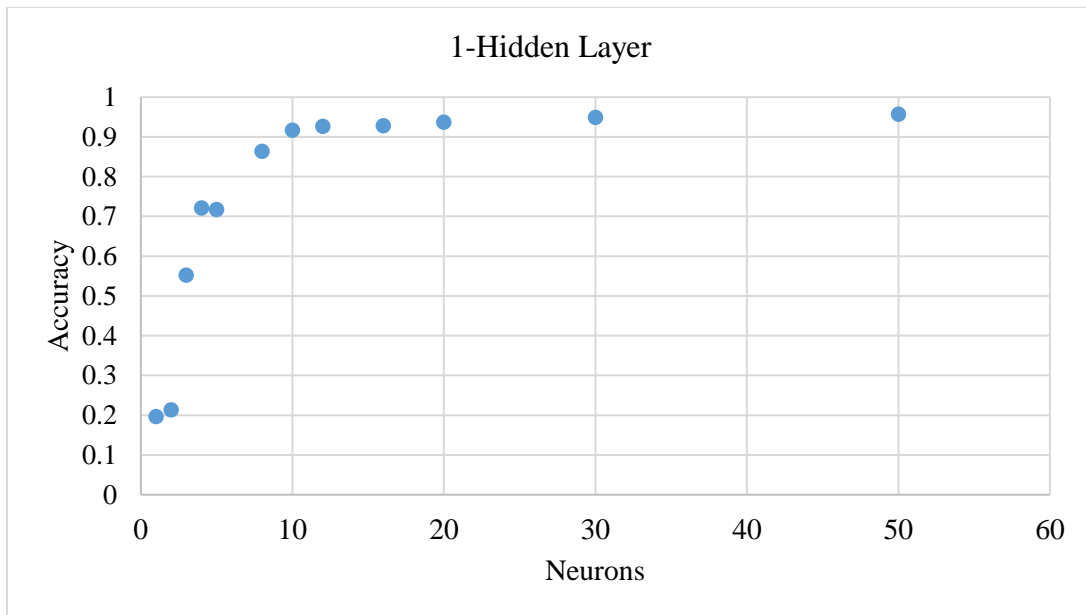


Figure 2.4

A graph comparing accuracy of different networks with 1 hidden-layer of various numbers of neurons.

Neurons	Accuracy
50	0.9568
30	0.9489
20	0.9366
16	0.9277
12	0.9264
10	0.9167
8	0.8639
5	0.7176
4	0.7213
3	0.552
2	0.2132
1	0.1961

Table 2.1

The data corresponding to graph in figure 2.4.

The results of a network using 1-hidden layer show that as you increase the number of the neurons in the middle layer from 1 up, the output increases exponentially and asymptotically approaches a value of 1.0.

When using 2-hidden layers, the following networks were tested; [784, 50, 50, 10], [784, 30, 30, 10], [784, 20, 20, 10], [784, 16, 16, 10], [784, 12, 12, 10], [784, 10, 10, 10], [784, 8, 8, 10], [784, 5, 5, 10], [784, 4, 4, 10], [784, 3, 3, 10], [784, 2, 2, 10], [784, 1, 1, 10]. Again, the second and third numbers show the value of neurons used in the first and second hidden layers. The results are shown on the next page in figure 2.5.

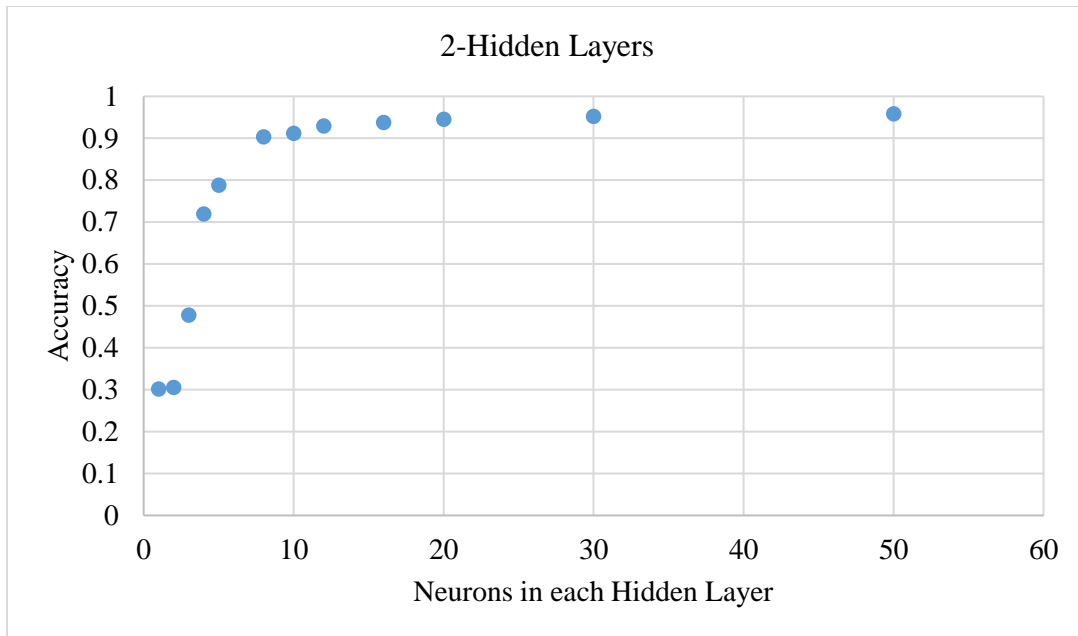


Figure 2.5

A graph comparing accuracy of different networks with 2 hidden-layers of various numbers of neurons.

Hidden Layer 1	Hidden Layer 2	Accuracy
50	50	0.9582
30	30	0.952
20	20	0.9453
16	16	0.9379
12	12	0.9296
10	10	0.9121
8	8	0.9032
5	5	0.7882
4	4	0.7194
3	3	0.4781
2	2	0.3055
1	1	0.3019

Table 2.2

The data corresponding to graph in figure 2.5.

The digit recognition accuracy results when using two hidden layers of the same size showed similar results to using one hidden layer. As you can see above, the accuracy increases in an exponential fashion when you add more neurons to each layer, then

asymptotically approaches near 100% accuracy. Below I have included a graph comparing the digit recognition accuracy of the one hidden-layer and two hidden-layer networks described above. As seen for the results, the two-hidden layer networks edge out the single hidden-layer networks.

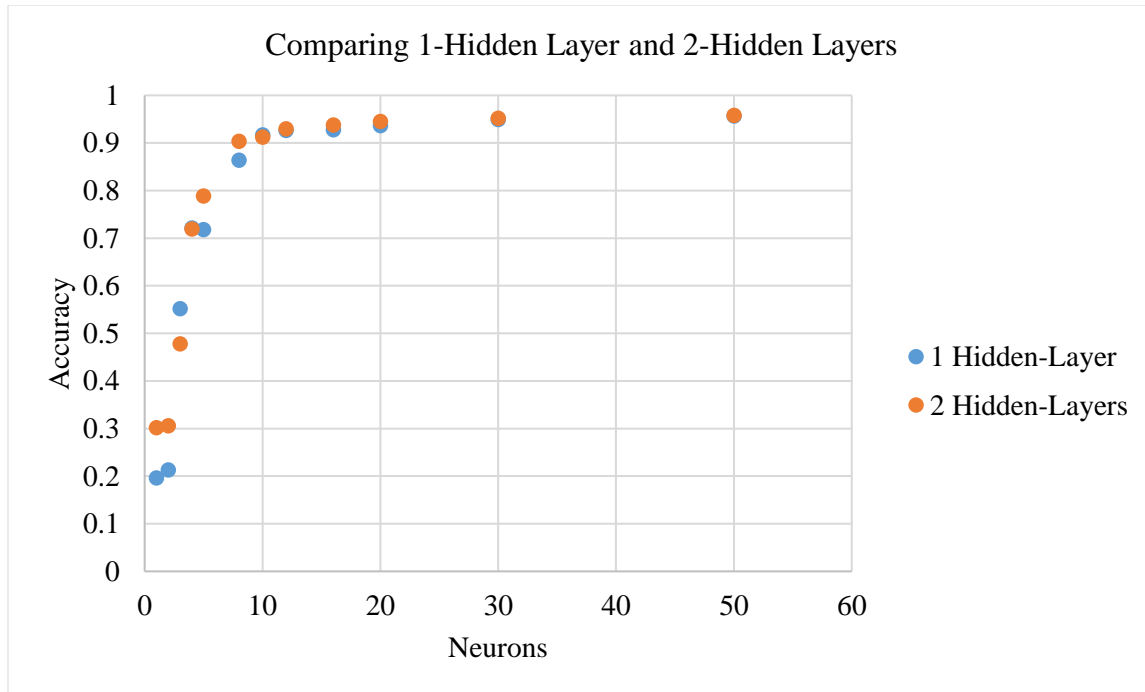


Figure 2.6

A graph comparing networks with a single hidden layer and two hidden layers.

For networks that have 10 neurons or greater in the hidden-layer(s), the difference in accuracy is especially small. This was taken into consideration when deciding the final network design.

2.5.2 Adjusting Epoch, mini_batch_size, and Learning Rate

With any network design, you can adjust the Epoch, mini_batch_size, and learning rate when training the network. These values are important to how quickly the network's weights and biases can be trained to the highest attainable digit recognition accuracy.

For my testing of these values, I used a network with 784 input-layer neurons, 12 hidden-layer neurons, and 10 output-layer neurons. I then set the Epoch = 20, learning rate = 3.0, then varied the mini_batch_size. The mini_batch_size is used to set the number size of the batches that are used to train the weights and biases. The result are shown below.

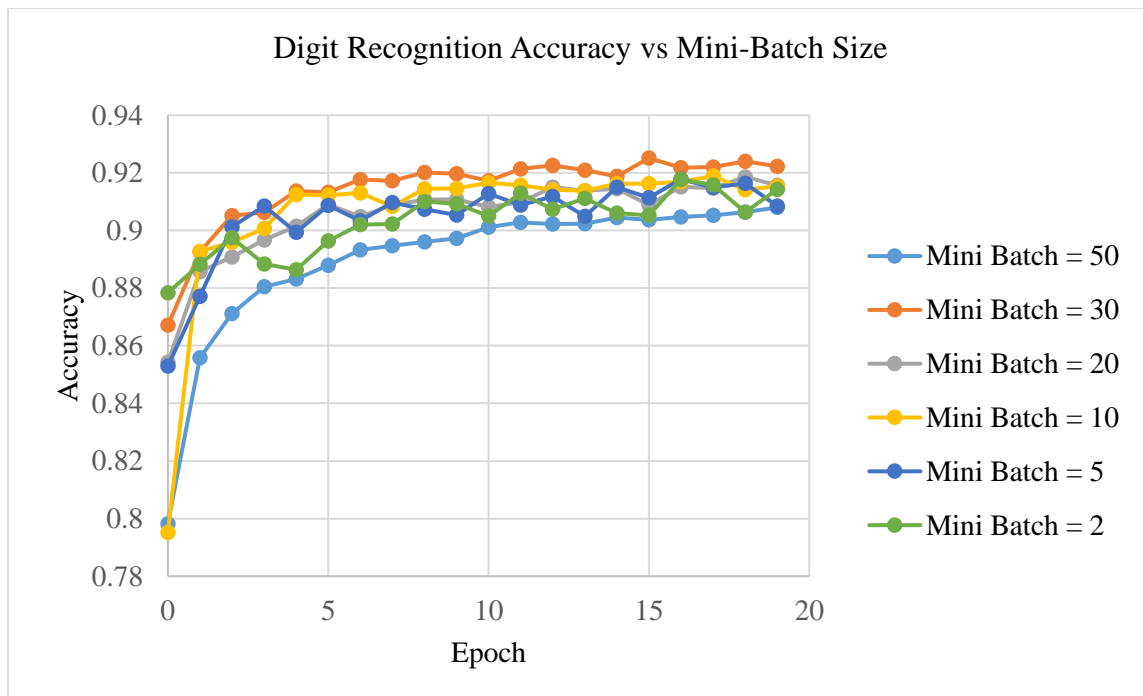


Figure 2.7

A graph comparing the digit recognition accuracy with the mini-batch size.

The results from changing the mini_batch_size showed that changes had a small effect on the overall digit recognition accuracy of the network after 20 epochs. Taking the results from above, I then tested changing the learning rate. For this case the values of mini_batch_size = 30 and epoch = 20.

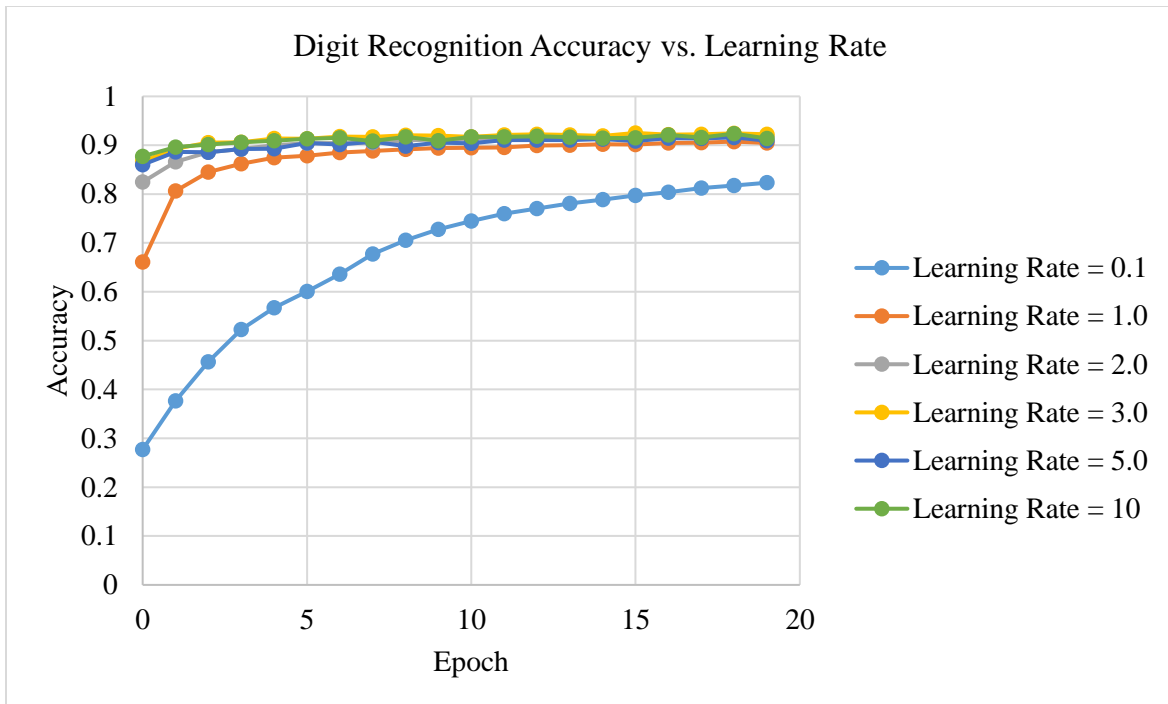


Figure 2.8

A graph comparing the digit recognition accuracy with the learning rate.

The results here show that by changing the learning rate, the number of epochs it takes for the network to reach the perceived ‘highest accuracy level’ of around 92%, is mainly what is affected here. For these results there is virtually no discernible difference in the performance of training for learning rates between 2.0 – 10.

One final test was conducted using four different combinations across 60 epochs. The combinations used here were, mini_batch_size = 10 & learning rate = 3.0, mini_batch_size = 10 & learning rate = 0.1, mini_batch_size = 30 & learning rate = 3.0, and finally mini_batch_size = 30 & learning rate = 3.0. The results of this test are shown on the next page.

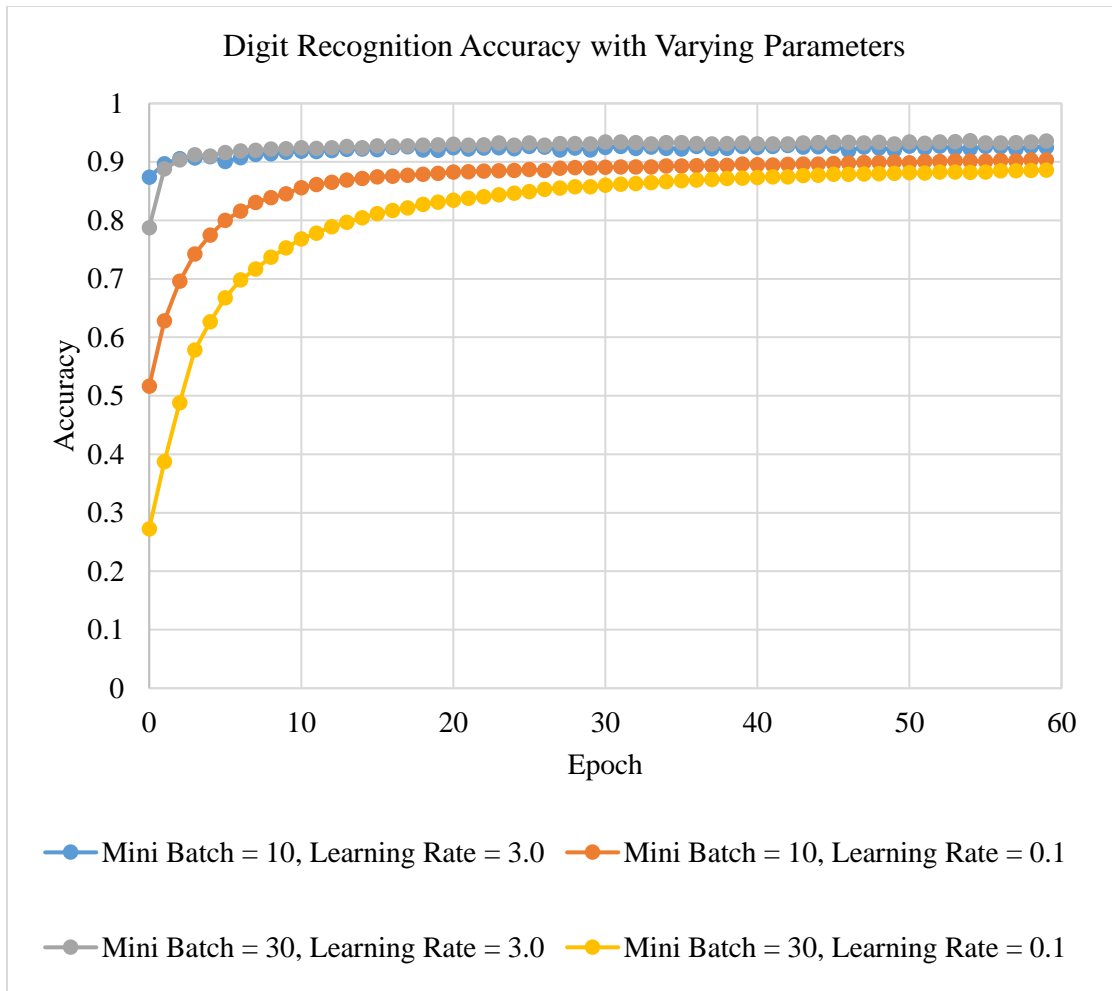


Figure 2.9

A graph comparing the digit recognition accuracy with varying parameters.

By testing these varying parameters, we are able to figure out which combination will lead to the most accurate set of weights and biases in the shortest amount of time. In my case, since the biases and weights are only going to be generated one time, then used in the network after that point, I can train the network for many more epoch, so that the accuracy can be as high as possible.

2.6 Final Network Design

As I stated before, when choosing the design of the network there were certain things that had to be taken into consideration. We wanted to have an accuracy that would be greater than 90%, preferably a little higher, as well as simple as a design as possible. This would keep the resource and power costs down.

The first thing that would need to be figured out is how many hidden layers would be needed. As shown in section 2.5.1, various networks with one and two hidden layers were compared. These results showed that networks with two hidden layers performed better than networks with a single hidden layer, but this increase was less than 1% in the case of the higher performing networks.

Once I had decided to only include a single hidden layer, I would now need to decide how many neurons would be in that layer. Again, the goal was to have greater than 90% accuracy, so I knew I would need at least 10 neurons in the hidden layer. A big consideration here was the need to reduce the number of neurons in order to make the design simpler in hardware. Each neuron would have to take as an input the 784 input neuron values multiplied by their respective weights. Each additional neuron would either add many floating-point adders, or additional cycles of latency to the system. Because of this tradeoff, I decided to go with 12 neurons in the hidden layer. This decision was made because at this number of neurons, each additional neuron does not add that much to the overall accuracy of the system. The final design of the network is shown in figure 2.10.

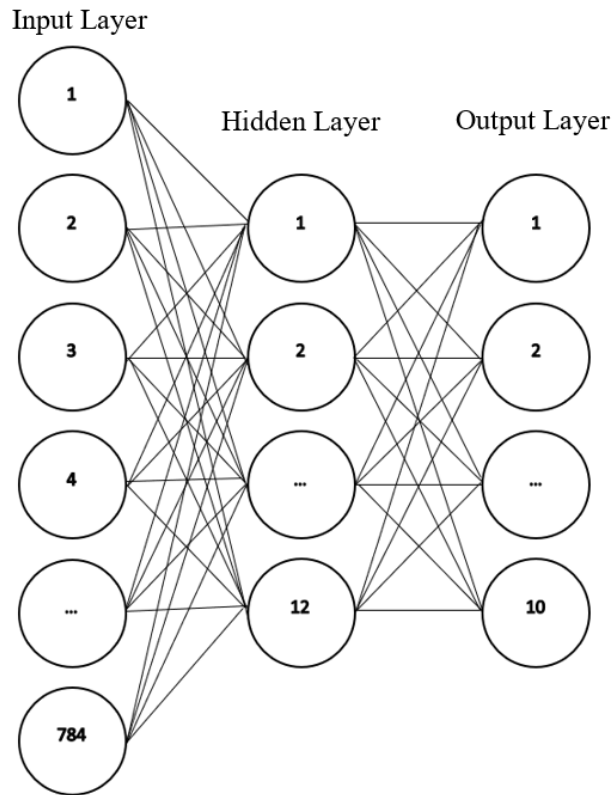


Figure 2.10
A figure showing the final network design.

Once a network design had been chosen, I could run the python program again to generate the final weights and biases that would be used. For this task, I used the techniques that were described in section 2.5.2, with Epoch = 60, mini_batch_size = 30, and Learning Rate = 3.0. There is a random nature to these numbers, so I ran this multiple times until I had a set of weights and biases that gave me an accuracy of 93.25%.

CHAPTER 3:

SOFTWARE IMPLEMENTATION

3.1 Matlab Implementation

Once a design had been chosen, the next step was to prove the approximate design with single precision floating-point numbers. This step is important in ensuring the weights and biases trained in Python work in my approximate design.

The implementation of the design begins with the weights and biases that were generated in Python. It is important to understand how the weights and biases are set up so that they can be correctly used in the network. The setup of weights0.csv is shown below. For this file we have a file with 9,408 single precision weights. These weights are distributed in 12 rows and 784 columns. Each row represents the weights associated with one of the 12 neurons in the hidden-layer.

w_{1_1}	w_{1_2}	w_{1_3}	...	$w_{1_{783}}$	$w_{1_{784}}$
w_{2_1}	w_{2_2}	w_{2_3}	...	$w_{2_{783}}$	$w_{2_{784}}$
w_{3_1}	w_{3_2}	w_{3_3}	...	$w_{3_{783}}$	$w_{3_{784}}$
...
w_{11_1}	w_{11_2}	w_{11_3}	...	$w_{11_{783}}$	$w_{11_{784}}$
w_{12_1}	w_{12_2}	w_{12_3}	...	$w_{12_{783}}$	$w_{12_{784}}$

Table 3.1
A layout of the weights of the first hidden layer.

The input image input into the network is a 28-pixel \times 28-pixel grayscale image. The way that these pixels are numbered is shown in table 3.2 on the next page.

p ₁	p ₂	p ₃	p ₄	...	p ₂₅	p ₂₆	p ₂₇	p ₂₈
p ₂₉	p ₃₀	p ₃₁	p ₃₂	...	p ₅₃	p ₅₄	p ₅₅	p ₅₆
p ₅₇	p ₅₈	p ₅₉	p ₆₀	...	p ₈₁	p ₈₂	p ₈₃	p ₈₄
...
p ₇₀₁	p ₇₀₂	p ₇₀₃	p ₇₀₄	...	p ₇₂₅	p ₇₂₆	p ₇₂₇	p ₇₂₈
p ₇₂₉	p ₇₃₀	p ₇₃₁	p ₇₃₂	...	p ₇₅₃	p ₇₅₄	p ₇₅₅	p ₇₅₆
p ₇₅₇	p ₇₅₈	p ₇₅₉	p ₇₆₀	...	p ₇₈₁	p ₇₈₂	p ₇₈₃	p ₇₈₄

Table 3.2

A layout of the input pixels of the first hidden layer.

For the first hidden layer, in the file biases0.csv, there are 12 biases, one for each neuron. The setup of the file biases0.csv is shown below.

b ₁
b ₂
b ₃
...
b ₁₀
b ₁₁
b ₁₂

Table 3.3

A layout of the biases of the first hidden layer.

The output for the hidden-layer neurons is then:

$$output(x) = \frac{1}{1 + \exp(-\sum_{i=1}^{784} wx_i * p_i - b_x)}$$

where in the above equation, i , refers to the input neuron, and the corresponding weight for that input, and x refers to the hidden layer neuron (1-12) that the output is being calculated for.

Once all 12 of the outputs of the hidden layer neurons are found, these values act as inputs to the 10 output layer neurons. A similar process in finding the outputs takes place as for the first hidden layer neurons, except this time we are using weights1.csv, and biases1.csv.

The file weights1.csv is laid out the same as weights0.csv, except this time there are 10 rows for the output neurons, and 12 columns for the inputs. An example of how the file is setup is shown below.

w1 ₁	w1 ₂	w1 ₃	...	w1 ₁₁	w1 ₁₂
w2 ₁	w2 ₂	w2 ₃	...	w2 ₁₁	w2 ₁₂
w3 ₁	w3 ₂	w3 ₃	...	w3 ₁₁	w3 ₁₂
...
w9 ₁	w9 ₂	w9 ₃	...	w9 ₁₁	w9 ₁₂
w10 ₁	w10 ₂	w10 ₃	...	w10 ₁₁	w10 ₁₂

Table 3.4
An example layout of the weights1.csv file.

The file biases1.csv is similar to biases0.csv, except there are only 10 values (one for each output neuron). An example of how this file is setup is shown below.

b ₁
b ₂
b ₃
...
b ₈
b ₉
b ₁₀

Table 3.5
An example layout output neuron biases.

The output for the output layer neurons is then:

$$output(x) = \frac{1}{1 + \exp(-\sum_{i=1}^{12} wx_i * hidden_out_i - b_x)}$$

where in the above equation, i , refers to the hidden layer neuron that is multiplied with corresponding, output of that neuron, hidden_out. The value, x , refers to the output layer neuron (1-10) that the output is being calculated for.

The way the whole network fits together is an image is input into the network. These 784 input values are used with the corresponding 9,408 weights, and 12 biases to

find the 12 outputs of the hidden layer neurons. The outputs of the hidden layer are then used with the corresponding 120 weights, and 10 biases to find the 10 outputs of the hidden layer neurons. Once this entire process has completed, the network has processed and classified one image.

The way the results are extrapolated from the neurons is simple. The output of each neuron is a value between 0 and 1.0. For each image, the network should converge on a single digit value. So the results for one neuron should be very close to 1.0 while the other values should be very close to 0. The neurons are assigned such that $\text{output}(1) = 0$, $\text{output}(2) = 1$, ..., $\text{output}(10) = 9$. An example output when running the program can be seen below.

```
The outputted value for 0 = 0.0000000070
The answer is 0
The outputted value for 1 = 0.0001626174
The answer is 1
The outputted value for 2 = 0.0000095044
The outputted value for 3 = 0.0000000021
The outputted value for 4 = 0.0000001204
The outputted value for 5 = 0.0000202689
The outputted value for 6 = 0.9999223948
The answer is 6
The outputted value for 7 = 0.0000013408
The outputted value for 8 = 0.0000000015
The outputted value for 9 = 0.0000022672
```

Figure 3.1

An example output when running the program.

The way the program works is it goes through each value of the outputs (0-9) sequentially, and if the current value is greater than the previous, it sets the value for variable 'answer' equal to that number. That is why we see the output 'The answer is 1', when that number is processed, because the value for 1 (0.0001626174), is greater than the value for 0 (0.000000007). We can see for the number 6, we have a value of

0.9999223948. Because this number is the highest, we can determine that the network has processed the input image, and classified it as a '6'.

3.2 Matlab Results

The implementation described above was run in Matlab using IEEE-754 double (64-bits), single (32-bits), and half (16-bits) precisions. The network was tested using the 10,000 MNIST test images. These are the same 10,000 images that were used to test the accuracy of the network in the Python program, and were not used in training the weights and biases.

In the first test, for each input image, all 10 of the outputs were compared and the highest value was determined to be the result. For instance, if we have the outputs shown below. The answer would be 3 even though the number for 3 (0.6894786954) isn't as strong as it could be (0.9999).

```
The outputted value for 0 = 0.0000003561
The answer is 0
The outputted value for 1 = 0.0000059766
The answer is 1
The outputted value for 2 = 0.0198363531
The answer is 2
The outputted value for 3 = 0.6894786954
The answer is 3
The outputted value for 4 = 0.0000000649
The outputted value for 5 = 0.0012522331
The outputted value for 6 = 0.0000001147
The outputted value for 7 = 0.0000000442
The outputted value for 8 = 0.0000994121
The outputted value for 9 = 0.0000465117
```

Figure 3.2

An example output when running the program with an input of 3.

This test ran for all 10,000 of the MNIST test images. The results below show how many images were correctly identified for half, single, and double precision data-types.

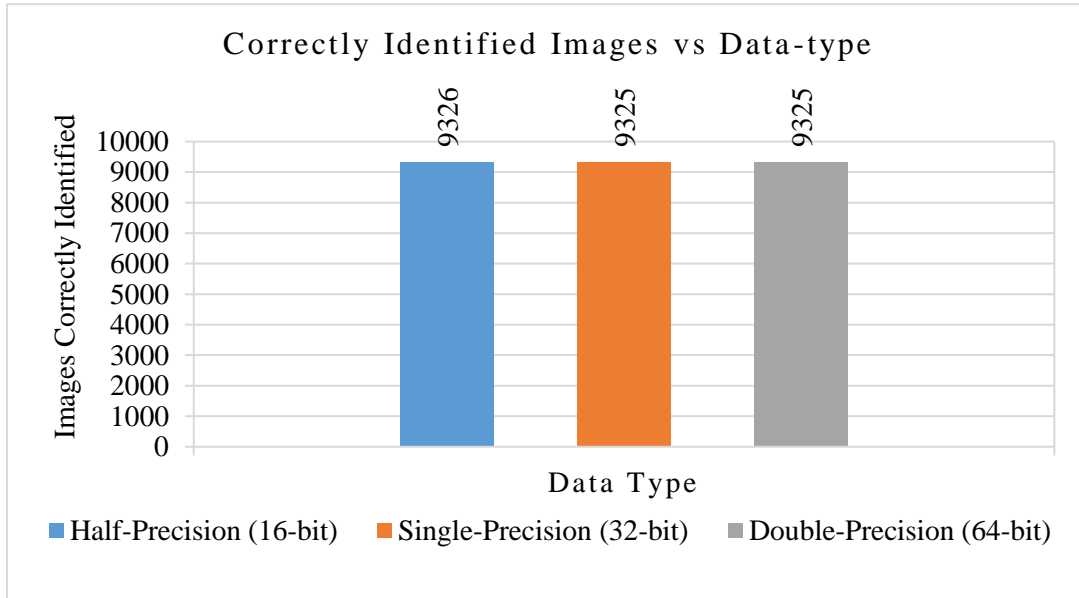


Figure 3.3
Comparison of half, single, and double precision results in Matlab.

With each correct identification the value of the number that the network classified could be any value from 0 - 1.0. The number that is output from each neuron can be described as the 'strength' of the result. With a value of 1.0 meaning the network has classified that as being the result with the highest probability, and a value of 0 meaning that it has the lowest chance of being that value. In figure 3.4 is a graph that shows the 'strength' of the images that were classified correctly.

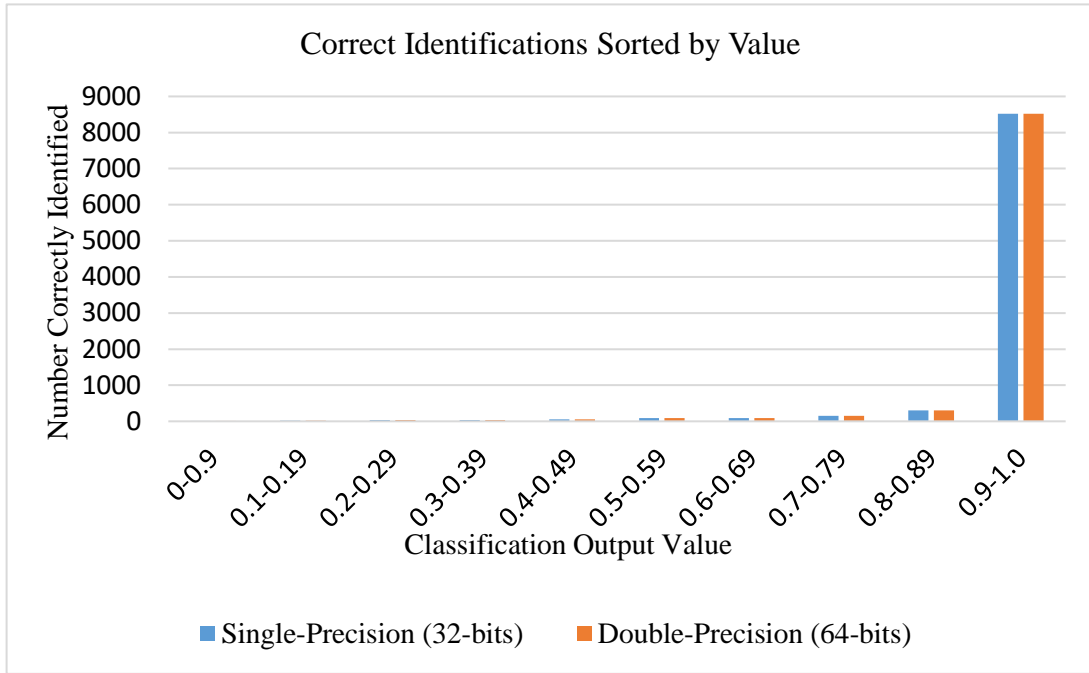
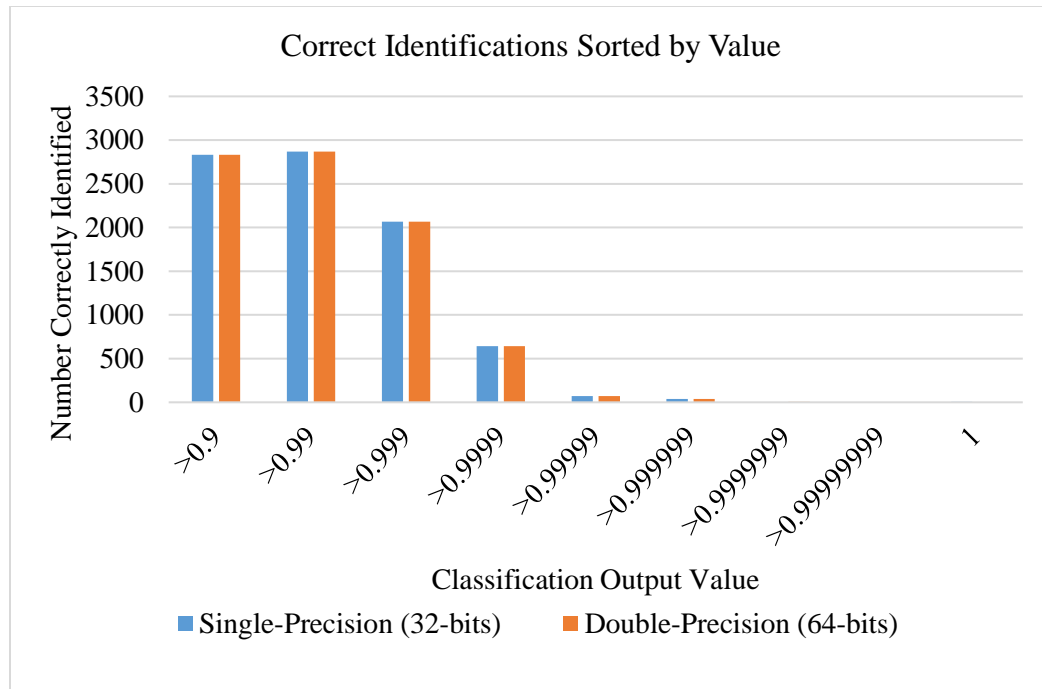


Figure 3.4
Comparison of the ‘strength’ of single and double precision data-types.

	0-0.9	0.1-0.19	0.2-0.29	0.3-0.39	0.4-0.49	0.5-0.59	0.6-0.69	0.7-0.79	0.8-0.89	0.9-1.0
Single-Precision (32-bits)	14	19	36	36	57	91	89	155	304	8524
Double-Precision (64-bits)	14	19	36	36	57	91	89	155	304	8524

Table 3.6
Corresponding data used for comparison of ‘strength’ of single and double precision datatypes.

From this data, we can see that for both single and double precision data types there are 8,524 out of 10,000 images correctly identified with a ‘strength’ of 0.9 or above. This data was examined further to find any differences between results with these data types. The graph below looks at the 8,524 correctly identified results, then breaks this data down further into different ranges.



*Figure 3.5
Further comparison of the ‘strength’ greater than 0.9 of single and double precision data-types.*

	>0.9	>0.99	>0.999	>0.9999	>0.99999	>0.999999	>0.9999999	>0.99999999	1
Single-Precision (32-bits)	2832	2067	644	71	38	0	0	0	5
Double-Precision (64-bits)	2832	2067	643	72	38	4	1	0	0

*Table 3.7
Corresponding data used for comparison of ‘strength’ greater than 0.9 of single and double precision datatypes.*

The data in the above table and graph can be broken down such that the first column represents which of the values are >0.9 and ≤ 0.99 , the second column represents values >0.99 and ≤ 0.999 , and on. Then the second to last column is values >0.999999999 and <1 , and finally the last column represents values that are equal to 1. From this data, you can see the main difference is that when using single-precision data types, there are 5 results that have a value of 1.0, meanwhile when using double precision, there are no results that have a value of 1.0. I will go into more detail interpreting these results in the next section.

3.3 Accuracy Comparison

The accuracy of the Matlab implementation of the neural network can be compared with the Python results obtained when training the network. This is because the network has the same weights and biases, and we are using the same 10,000 MNIST test images in order to test the network. The comparison of results is shown below.

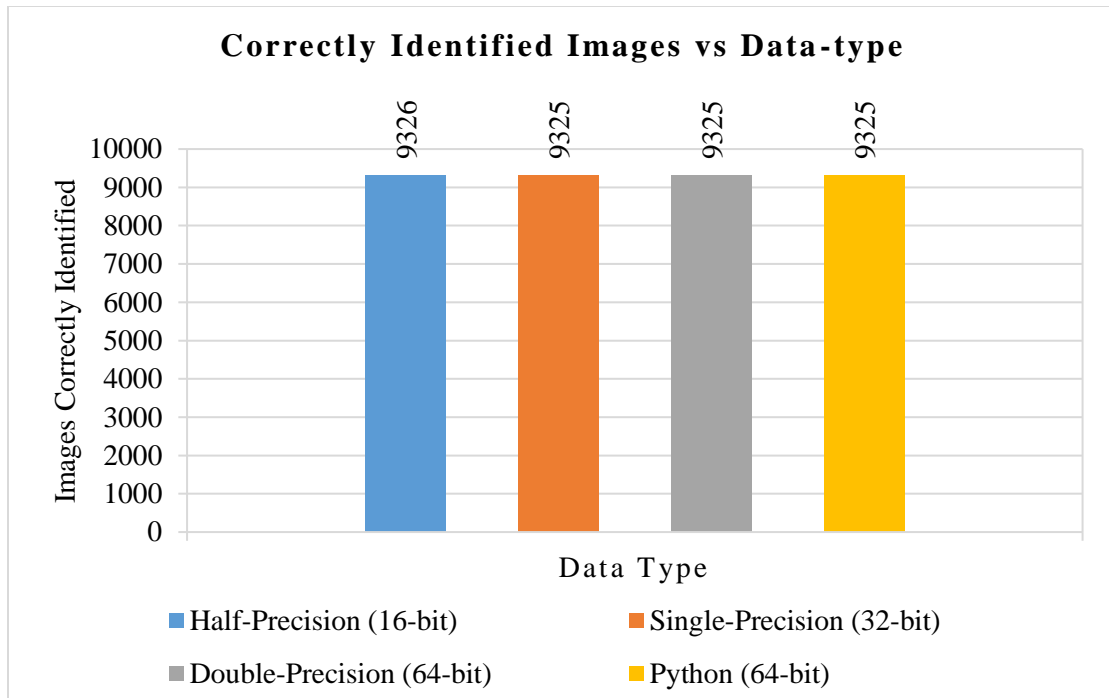


Figure 3.6

Comparison of digit recognition accuracy of Matlab implementations with Python results.

In Python, the weights and biases are created using 64-bit floating point numbers, therefore all the weights and biases are 64-bit, and must be converted to 32-bit and 16-bit for single and half precision. From this data we can see that the Python results, single, and double precision results are all the same. Surprisingly, the half-precision results in Matlab correctly identified 9326 values out of the 10,000. This is one better than all other results and is unexpected. Another set of weights and biases for the network was created

in Python, this time correctly identifying 9267/10000 images. These weights and biases were then run in Matlab, where all half, single, and double precision implementations correctly identified 9267/10000 images. Therefore the discrepancy seen in the results above is most likely associated with a slight change in internal numbers when changing to lower granularity at half-precision.

Another difference in the data that I want to cover in more detail here is the reason why for the single-precision results, there are 5 correctly identified results that are equal to 1.0, and no results >0.9999999 that are not equal to 1.0. Meanwhile for double precision, there are no results equal to 1.0 and 5 results >0.9999999 that are not equal to 1.0.

This difference can be attributed to the fact that doubles are more precise than singles, just as the name would suggest. For single precision numbers, any values with precision of 0.99999998 or greater are not able to be represented. These numbers are then converted to 1.0 when represented in single-precision floating point. For double-precision numbers meanwhile, these numbers are able to be represented. This is why for double precision values, there are 5 numbers that are >0.9999999 and <1.0 , and for single precision there are no values and 5 that are equal to 1.0.

Overall, the results have proven the proposed network design for half, single, and double precision data types. Comparing all of these results with the initial Python results, we can see that there is no discernible difference when using different precision data types. Once I had these results I could start the process of taking this design and implementing it in hardware.

CHAPTER 4:

HARDWARE IMPLEMENTATION

4.1 Hardware Design Architecture

When thinking about implementation in hardware, the first thing was to break the algorithm down by the operations that would need to happen. We can take equation for the output of the hidden layer neurons in section 3.1, and write it as the following:

$$output(x) = \frac{1}{1 + \exp(-K - b_x)}$$

$$\text{where } K = \sum_{i=1}^{784} wx_i * p_i$$

This allows us to break the problem of finding the output of a hidden layer neuron down into two parts. First, we need to multiply all 784 input pixels by their corresponding weights, then sum those values into one result. Second, we need to take that summed value and plug it into the sigmoid function. This involves five different operations; first taking the negative value of the summed result, then subtracting bias, then taking exp to that value, then adding a value of 1, then finally taking the reciprocal.

A visual representation of the first part of this process is shown below.

Clock Cycle:	1	2	3	4	5	6	7	8	9	10	11
Pixel1	Multiplier1										
Weight x 1		Adder1_1									
Pixel2	Multiplier2										
Weight x 2			Adder2_1								
Pixel3	Multiplier3			Adder3_1							
Weights x 3		Adder1_2			Adder4_1						
Pixel4	Multiplier4					Adder5_1					
Weights x 4							Adder6_1				
...								Adder7_1			
									Adder8_1		
										Adder9_1	
											Adder10_1
...											
Pixel781	Multiplier781										
Weight x 781		Adder1_391									
Pixel782	Multiplier782			Adder3_98							
Weight x 782			Adder2_196								
Pixel783	Multiplier783										
Weights x 783		Adder1_392									
Pixel784	Multiplier784										
Weights x 784											

Figure 4.1

A visual representation of the multipliers and adders going into the first layer hidden neurons.

In this representation we are assuming that each operation has a latency of one clock cycle. We can see that in the first clock cycle, we are using 784 multipliers to multiply the input pixels by their correct weights. There are then 10 stages of cascading adders to sum the results up into one final value. This value is then used as the input for the second part in finding the output of the hidden layer neurons. This stage is shown below.

Clock Cycle:	12	13	14	15	16	17
Adder10_1_result	Negative	sub bias	exp	add 1.0	Recip	Neuron_final_result

Figure 4.2

A visual representation of the results from the multipliers and adders going into the final operations of the neuron.

In the above image, we can see the second stage where we are taking the negative value of our result, subtracting the bias, taking that answer to exp, adding 1, then finally taking the reciprocal. This whole process takes another 5 clock cycles, if we assume a latency of one cycle for every operation. Putting the two steps together, it takes a total of 16 clock cycles in order to find the output of a the hidden layer neuron.

The second stage in the hardware involves taking the outputs of the hidden-layer neurons, then plugging those values into the following equation:

$$final_output(x) = \frac{1}{1 + \exp(-K - b_x)}$$

$$\text{Where } K = \sum_{i=1}^{12} wx_i * hidden_output_i$$

This stage is implemented in a similar manner as the first stage, but instead of 784 inputs, we only have 12 this time. A visual representation is shown below.

Clock Cycle:	1	2	3	4	5	6	7	8	9	10	11
Hidden_out1	Multiplier1										
Weight_x_1		Adder1_1									
Hidden_out2	Multiplier2										
Weight_x_2			Adder2_1								
Hidden_out3	Multiplier3										
Weights_x_3		Adder1_2		Adder3_1							
Hidden_out4	Multiplier4										
Weights_x_4					Adder4_1	Negative	sub bias	exp	add 1.0	Recip	
...							Neuron_final_result
Hidden_out9	Multiplier9										
Weight_x_9		Adder1_5		Register2_3							
Hidden_out10	Multiplier10										
Weight_x_10			Adder2_3								
Hidden_out11	Multiplier11										
Weights_x_11		Adder1_6									
Hidden_out12	Multiplier12										
Weights_x_12											

Figure 4.3

A visual representation of the operations of a single output layer neuron.

In this stage, because there are only the 12 inputs, it takes only takes 4 layers of adders to sum the multiplication results, this allows the entire process of calculating the result of the final layer neurons to take only 10 clock cycles.

The design described above describes the components needed for one neuron in the hidden layer, and one neuron in the output layer. This design can be implemented for every neuron in the hidden layer, as well as every neuron in the output layer by repeating the design 12 and 10 times respectively. You can also pipeline the design by in the first clock cycle use the data for the first hidden layer neuron, on second clock cycle use data for second layer neuron, etc. This adds complexity because to begin calculations for the output layer neurons, you must first know the results from all the hidden layer neurons. I have compared the latency and resource usage for these two approaches below.

	Latency (clock cycles)	Multipliers	Adders	Subtractors	Exponential	Reciprocal
Non-pipelined	26	9528	9528	44	22	22
Pipelined	48	796	796	4	2	2

Table 4.1

A comparison of the resources needed for a pipelined vs non-pipelined design.

When comparing these two approaches, you can see that even though the non-pipelined approach is nearly twice the speed of the pipelined approach, it uses nearly 12 times the resources. Because of this issue, I decided to go with a design that would be pipelined, in order to lower my resources usage. I still had issue with the fact that the number of multipliers and adders was so high.

With this problem in mind, I decided to break down the multiplication of the first layer weights and inputs into 8 different steps. This portion of the design accounts for 784/796 multipliers and 783/796 adders, if I break this down into multiple parts, I can lower the overall resource utilization tremendously.

This design keeps the same pipelined structure for the output level neurons, but for the hidden layer neurons, instead of 784 multipliers followed by adders, we will only have 98. This design is shown below.

Clock Cycle:	1	2	3	4	5	6	7	8
Pixel1	Multiplier1							
Weight_x_1		Adder1_1						
Pixel2	Multiplier2							
Weight_x_2			Adder2_1					
Pixel3	Multiplier3			Adder3_1				
Weights_x_3		Adder1_2			Adder4_1			
Pixel4	Multiplier4					Adder5_1		
Weights_x_4							Adder6_1	
...		Adder7_1
Pixel95	Multiplier95						Adder6_2	
Weight_x_95		Adder1_48			Adder4_6	Adder5_3		
Pixel96	Multiplier96			Adder3_12				
Weight_x_96			Adder2_24					
Pixel97	Multiplier97							
Weights_x_97		Adder1_49						
Pixel98	Multiplier98							
Weights_x_98								

Figure 4.4

A visual representation of the first stage multiplication with 98 multipliers.

With this design, we are able to process 98 inputs with each iteration. So we will have to have 8 iterations in order to process all the inputs for a neuron. The way the design works is like a pipeline within a pipeline. For the first input neuron, we have to run the above 98 multiplier adder sequence 8 times. As the results from each eighth of the multiplications come through they are added together in an accumulator. An idea of how the accumulator fits into the output of the hidden layer neurons is shown below.

Clock Cycle:	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Adder7_1_result	Accum1	Accum2	Accum3	Accum4	Accum5	Accum6	Accum7	Accum8	Negative	sub bias	exp	add 1.0	Recip	Neuron_final_result

Figure 4.5

A visual representation of the accumulation and final operations used after the 98 multipliers.

Looking at the above image, you can see at cycle 9 the first value from the multiplier-adder comes in. For 8 cycles these values are accumulated to get one final value for all 784 inputs. Once this value has been found, the result is just sent into the same sequence, of taking the negative, subtracting bias, taking exponential, adding 1.0, then taking the reciprocal. This entire process ends up taking 21 cycles to find the result. A comparison of the resource utilization and latency of this design compared to pipelined and non-pipelined is shown below.

	Latency (clock cycles)	Multipliers	Adders	Subtractors	Exponential	Reciprocal
Non-pipelined	26	9528	9528	44	22	22
Pipelined	48	796	796	4	2	2
Pipelined/98-mul	129	110	110	4	2	2

Table 4.2

A comparison of the resources needed for a pipelined, non-pipelined design, and pipelined with 98 multipliers designs.

We can see that by adding this change, we are using 13.8% the number of multipliers and adders as the fully-pipelined design and 1.15% the number of multipliers and adders as the non-pipelined design. This does come at a cost of a higher latency, but given that the overall number of cycles needed for the network to classify one image is only 129, this still remains very low and still feasible for real-time applications. Because of these results, the hardware design architecture was finalized as the fully-pipelined design with 98 multipliers for the first hidden-layer.

4.2 Final Hardware Architecture

The design and tradeoffs of the various non-pipelined, pipelined, and pipelined with 98 multiplier architectures were discussed at length in the previous section. In this section I will describe the components used in the final architecture, and how they are connected.

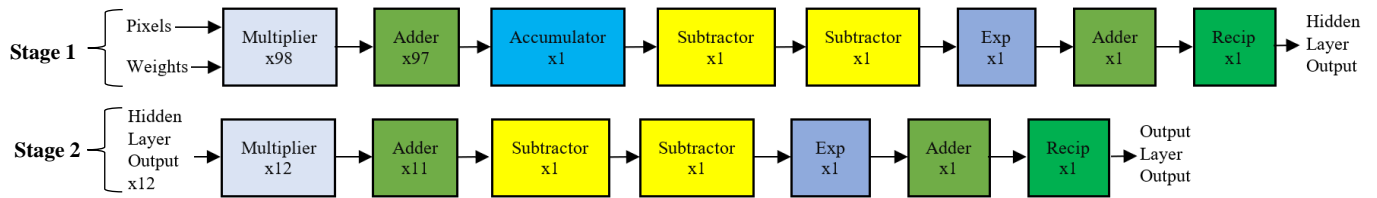


Figure 4.6

A visual representation of all the components of the design put together.

The above diagram shows the connection of the hardware components. Stage 1 is used for finding the output of the hidden layer neurons, and stage 2 is used for finding the output of the output layer neurons. In stage 1, the 98 multipliers and 97 cascading adders are used 8 times, feeding the result each time into the accumulator. This allows the network to process all 784 input pixels and their corresponding weights. Once the accumulator has accumulated all 8 summed values, the results propagate sequentially

through the rest of stage 1. Stage 1 is pipelined for all 12 of the hidden-layer neurons. Once all 12 of the stage 1 outputs have been calculated, stage 2 processes the results of the output layer neurons sequentially. Stage 2 is again pipelined for all 10 output neurons.

4.3 Timing and RTL Design

4.3.1 How the Counters are Used

Given the pipelined nature of the design, timing became very important in making sure that at any given moment the input pixels, weights, and biases were correct. The way that this problem was solved was by using six different counters that would serve as references for different inputs and outputs within the design.

- Counter 1: Used to control the input pixels and weights that are being input into the 98 multipliers of stage 1.
 - Counts from 0-9. Increments counters every clock cycle. Resets to 0 when it reaches a value of 9.
 - Enabled when the entire network is enabled.
 - Inputs/weights change from 0-7, then two extra cycles allow for accumulator to be reset in between neurons.
- Counter 2: Used to control for which neuron the input weights in stage 1 will correspond to.
 - Counts from 0-11. Increments every time Counter 1 reaches 9 and resets to 0. Resets to 0 when it reaches a value of 11 and Counter 1 reaches value of 9.
 - Enabled when the entire network is enabled.
 - Changes the input neuron weights every time it is incremented, cover all hidden layer neurons (0-11).

- Counter 3: Used to control the when accumulator is used.
 - Counts from 0-9. Increments every clock cycle. Resets to 0 when it reaches a value of 9.
 - Enabled when Counter 1 = 8 and Counter 2 = 0.
 - Used to enable to accumulator when Counter 3 is enabled and less than 8, and to resets the accumulator when Counter 3 is greater than 7.
- Counter 4: Used to control when the output of the hidden layer neuron is valid.
 - Counts from 0-9. Increments every clock cycle. Resets to 0 when it reaches a value of 9.
 - Enabled when Counter 1 = 6 and Counter 2 = 1.
 - When Counter 4 = 5, the result is stored to a location in a buffer, dependent on the value of Counter 5.
- Counter 5: Used to tell to which neuron (0-11) a valid hidden layer neuron result is stored.
 - Counts from 0-11. Increments every time Counter 4 reaches 9 and resets to 0. Resets to 0 when it reaches a value of 11 and Counter 4 = 9.
 - Enabled when entire network is enabled.
 - Tells which valid result corresponds to which neuron (0-11) in the hidden layer results buffer.
- Counter 6: Used to tell when the output of the output layer neuron is valid.
 - Counts from 0-19. Increments every clock cycle. Resets to 0 when it reaches a value of 19.
 - Enabled when Counter 4 = 9 and Counter 5 = 11.
 - Signals when the output of the entire network is valid. When Counter 6 = 0xa, the result corresponds to output-neuron[0] value, Counter 6 = 0xb

corresponds to output-neuron[1] value, ..., Counter 6 = 0x13 corresponds to output-neuron[9] value.

A diagram showing all timers in line with tasks of the network is shown in figure 4.7 on the next page.

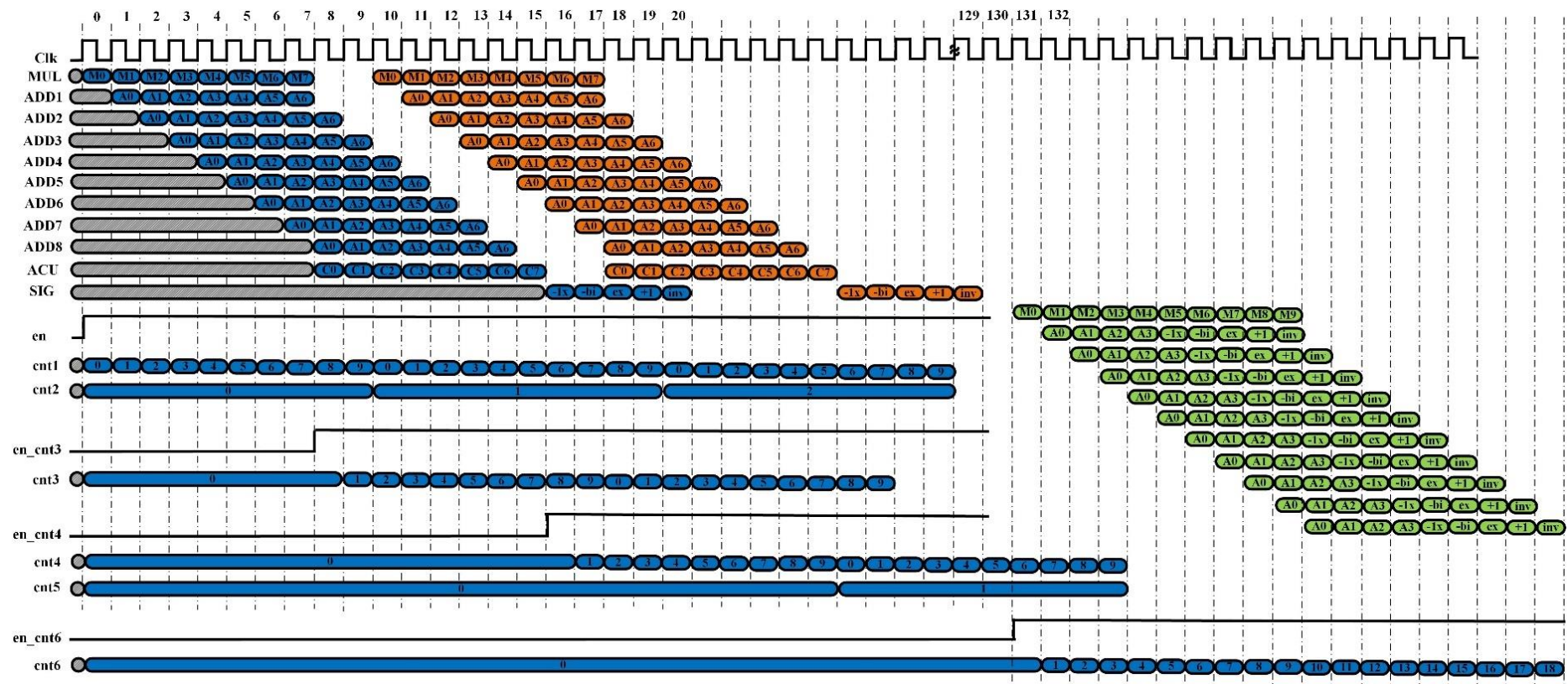


Figure 4.7
A visual diagram of the timing of operations aligned with counters.

In the above diagram, you can see the sequential nature of how the result from M0 feeds into ADD1, which then takes 7 clock cycles to add the 98 results together, then at clock cycle 8, the accumulator starts adding together the 8 results, then feeding them into the sigmoid operations. Underneath these operations, you can see how the counters line up and signal correctly for all the attributes listed above.

4.3.2 RTL Design

Due to the number of operational components needed for this network (adder, multiplier, exponential, etc.) as well as the fact that these operations needed to take place with floating point inputs, the decision was made to use Vivado IP. I will go over the implementation of each individual IP thoroughly in the next section. Therefore, in this section I will describe the general RTL design used to describe the enables and counters used.

A general enable was used when the entire network turns on. With this signal, we are allowing Cnt1, Cnt2, and Cnt5 to begin. Counter that have an enable specific to them have the structure shown below.

```
wire en_cntx_w, en_cntx;
reg en_cntx_r;
assign en_cntx_w = (CONDITION 1) & (CONDITION 2);
assign en_cntx = en_cntx_w ^ en_cntx_r;
always @(posedge clk or negedge rst) begin
    if (~rst) begin
        en_cntx_r <= 1'b0;
    end else begin
        en_cntx_r <= en_cntx;
    end
end
```

In this code snippet, 'x' corresponds to the counter that this enable is used for. The values for CONDITION 1 and CONDITION 2, are user defined and can include one or more conditions on which you would like the counter to be enabled. A simplified hardware translation of this design is shown below:

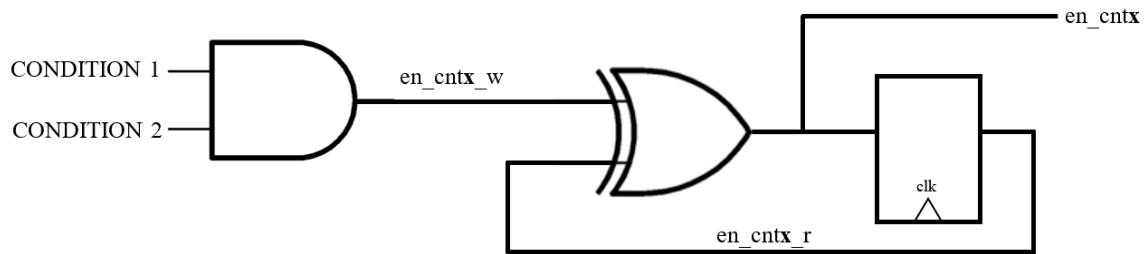


Figure 4.8
A simplified hardware translation of the enable design.

From this design, we can see that once, CONDITION 1 and CONDITION 2 have both been met, this gives en_cntx_w a value of 1, and en_cntx a value of 1. This then gives en_cntx_r a value of 1 on the next clock cycle. On the next clock cycle, the value of en_cntx_w should no longer be 1, and therefore the value of en_cntx continues to maintain a value of 1 indefinitely. This design is used to enable counters 3, 4, and 6.

The design for the counters is also very simple and similar to the design for the enables. A snippet of code used for Cnt1 is shown below.

```
assign nxt_cnt1 = (en & cnt1==4'd9) ? 4'd0 :
                  en ? (cnt1+4'd1) : cnt1;
always @(posedge clk or negedge rst) begin
  if (~rst) begin
    cnt1 <= 4'd0;
  end else begin
    cnt1 <= nxt_cnt1;
  end
end
```

In this counter design, at every posedge of the clk we are assigning the register Cnt1 to be the value of the wire nxt_cnt1. The value of nxt_cnt1 is determined by an assign statement where we are controlling the en, that is used, when cnt1 is reset to 0 ((en & cnt1 == 4'd9) ? 4'd0), and the rate at which we are increasing the counter (cnt1+4'd1). For each counter, the differences are in the assign statement where we are deciding when the counter is enabled, how to count, and when to reset.

4.4 Vivado IPs

As stated above, due to the number of components and the fact that they were floating point. The decision was made to use IPs designed by Vivado in order to represent the multiply, addition, subtraction, accumulation, exponential, and reciprocal operation. Since the Matlab results had shown no noticeable degradation in results when using lower precision numbers, the decision was made to use the lowest precision. However, half-precision inputs were not available for all the needed IPs, so the decision was made to use single-precision inputs for all floating-point IP.

4.4.1 Instantiating the IPs

From a project within Vivado the sources for the floating-point IP can be found by going to IP Catalog > Floating Point. Once you double click on 'Floating Point', a new window will pop up where you can specify all the attributes that you desire.

For each operation, there are slightly different options when you are describing the IP so I will include screenshots below for each IP. For every IP, the flow-control should be specified as 'non-blocking' with the latency set to '1', and the precision of inputs should be set to 'single'. These options are saying that the latency of each of our IPs will be equal to 1 clock cycle. This is a critical assumption made in our timing considerations, so this value must be 1 for the network to function properly. A description of the options for each component is show below. For each component, all options are to be left at default except for what is described below.

Floating-Point Adder: Component Name: fp_add

Operation Selection > Operation Selection: Add/Subtract

Operation Selection > Add/Subtract and FMA Operation options: Add

Precision of Inputs > A Precision Type: Single

Optimizations > DSP Slice Usage: Full Usage

Interface Options > Flow Control Options / Flow Control: Nonblocking

Interface Options > Latency and Rate Configuration / Use Maximum Latency:

Unchecked

Interface Options > Latency and Rate Configuration / Latency: 1

Floating-Point Subtractor: Component Name: fp_sub

Operation Selection > Operation Selection: Add/Subtract

Operation Selection > Add/Subtract and FMA Operation options: Sub

Precision of Inputs > A Precision Type: Single

Optimizations > DSP Slice Usage: Full Usage

Interface Options > Flow Control Options / Flow Control: Nonblocking

Interface Options > Latency and Rate Configuration / Use Maximum Latency:

Unchecked

Interface Options > Latency and Rate Configuration / Latency: 1

Floating-Point Accumulator: Component Name: fp_acc

Operation Selection > Operation Selection: Accumulator

Operation Selection > Add/Subtract and FMA Operation options: Add

Precision of Inputs > A Precision Type: Single

Optimizations > DSP Slice Usage: Full Usage

Interface Options > Flow Control Options / Flow Control: Nonblocking

Interface Options > Latency and Rate Configuration / Use Maximum Latency:

Unchecked

Interface Options > Latency and Rate Configuration / Latency: 1

Interface Options > Control Signals / ACLKEN: Checked

Interface Options > Control Signals / ARESETn: Checked

Floating-Point Multiplier: Component Name: fp_mul

Operation Selection > Operation Selection: Multiply

Precision of Inputs > A Precision Type: Single

Optimizations > DSP Slice Usage: Max Usage

Interface Options > Flow Control Options / Flow Control: Nonblocking

Interface Options > Latency and Rate Configuration / Use Maximum Latency:

Unchecked

Interface Options > Latency and Rate Configuration / Latency: 1

Floating-Point Exponential: Component Name: fp_exp

Operation Selection > Operation Selection: Exponential

Precision of Inputs > A Precision Type: Single

Optimizations > DSP Slice Usage: Full Usage

Interface Options > Flow Control Options / Flow Control: Nonblocking

Interface Options > Latency and Rate Configuration / Use Maximum Latency:

Unchecked

Interface Options > Latency and Rate Configuration / Latency: 1

Floating-Point Reciprocal: Component Name: fp_recip

Operation Selection > Operation Selection: Reciprocal

Precision of Inputs > A Precision Type: Single

Optimizations > DSP Slice Usage: Full Usage

Interface Options > Flow Control Options / Flow Control: Nonblocking

Interface Options > Latency and Rate Configuration / Use Maximum Latency:

Unchecked

Interface Options > Latency and Rate Configuration / Latency: 1

4.4.2 Distributed Memory Generator IP

In order to store the input pixels, weights, and biases, the distributed memory generator IP was used. The use of this IP allowed us to initialize data the input data in the form of COE files into these memory locations. This IP can be found by going to the IP Catalog > Distributed Memory Generator. Once you double-click on the distributed memory generator. The options selected are shown below.

Distributed Memory Generator IP: Component Name: Dependent on weights/biases/etc.

Memory config > Options / Depth: Dependent on weights/biases/etc.

Memory config > Options / Data Width: 32

Memory config > Memory type / Memory type: ROM

Port config > Input Options / Input Options: Non Registered

RST & Initialization > Load COE File / Coefficients File: Dependent on weights/biases/etc.

In the above description, the Component name, memory depth, and COE file are dependent on which specific memory component is being described. Overall, there are 98 components for the input pixels, 98 components for the first input weights, 12 input components for the second input weights, and 2 components for the two different biases.

For the input pixels, the names are pixels0, pixels1, pixels2, ..., pixels97. The depth is 16 for all pixels, and the COE files have names corresponding to the pixels (i.e. pixels0.coe for the pixels0 memory).

For the first input weights, the names are weights1_0, weights1_1, ..., weights1_97. The depth is 96 for all the weights, and the COE files have names corresponding to the weights (i.e. weights1_0.coe for the weights1_0 memory).

For the second input weights, the names are weights2_0, weights2_1, ..., weights2_11. The depth is 16 for all the weights, and the COE files have names corresponding to the weights (i.e. weights2_0.coe for the weights2_0 memory).

For the two input biases, the names are bias1 and bias2. The depth is 16 for both the biases, and the COE files have names corresponding the bias, bias1.coe for bias1 and bias2.coe for bias2.

4.4.3 How to Create COE files Using Matlab

In order to load the correct values into memory using the COE files, I had to take the CSV files for weights0, weights1, biases0, biases1, and input pixels and convert to single precision floating point binary values. An example of the COE file for biases0 is shown below.

```

1 MEMORY_INITIALIZATION_RADIX=2;
2 MEMORY_INITIALIZATION_VECTOR=
3 01000000010011011011100011000111,
4 0100000001010001000100001011000,
5 0011111000110110010111001010111,
6 101111110100000000111100110111,
7 10111111101001110100101100111101,
8 10111111101001010001000110100101,
9 01000000000110111110111010100011,
10 11000000001100011100001000010100,
11 10111110001000110000101011000000,
12 10111101101000010101011011001100,
13 00111111110001001001111011010100,
14 11000000011111111100101100000100,
15 00000000000000000000000000000000,
16 00000000000000000000000000000000,
17 00000000000000000000000000000000,
18 00000000000000000000000000000000;
```

Figure 4.9

An example COE file for the hidden layer biases.

In all the COE files, the first two lines are the same. These set the radix on which the COE file is read, and starts the vector of information. In this file, we have a Depth of

16, and a Width of 32. You can see this by noting that there are 16 lines of data, each of which is 32 bits wide. In this particular case we are representing the 12 biases for the hidden layer neurons. Therefore each of the first 12 values in this file are the single-precision binary value for the bias associated with each neuron. In the distributed memory generator IP, the smallest depth allowed is 16, and since we only have 12 values, the last 4 pieces of data have a value of 0.

In the file `weights1_coe.m` we are creating the 98 COE files that will be used in the initialization of the 98 memories used as inputs into the first stage multipliers. Because the value used must correspond to the values used in each of the 98 pixel input COE files created by `pixels_coe.m`, extra care must be taken. The way that the COE files for the hidden layer weights and input pixels is arranged is such that the first COE file has the first value, the second COE file has the second value, etc. In this arrangement, each weights COE file has 8 single precision values for the weights of each neuron, such that it has 96 values total when storing the weights for all the neurons. Each of the 98 pixel COE file has 8 values.

In Matlab I have created 5 files, `pixels_coe.m`, `weights1_coe.m`, `weights2_coe.m`, `biases1_coe.m`, and `biases2_coe.m`. These files can be run from the same path as the files `weights0.csv`, `weights1.csv`, `biases0.csv`, `biases1.csv`, `load_mnist.m`, and `t10k-images.idx3-ubyte` in order to generate all 210 required COE files.

4.5 Running the Project

In this section I will cover all the steps needed to create the project using the TCL script, running the synthesis, generating the waveform, and interpreting the results of the waveform.

4.5.1 Creating the Project Using TCL

In order to simplify the creation of this project I created a TCL script that can be used with Vivado's TCL Shell to create the project and add all the appropriate IP. The decision to make a TCL file to automate this process is due to the fact that there are 210 instances of the Distributed Memory Generator alone. To manually add each instance with the correct COE file into the project would take quite some time, while the TCL script can do this all automatically based on predefined values. In the TCL script I am creating the project based upon the xcku035-sfva784-1LV-I part, adding all the sources, instantiating all the floating point and distributed memory generator IPs, then launching the GUI. Once the GUI has been launched you are able to do the synthesis or simulation just like you would if you created the project manually.

4.5.2 Running the TCL Script

The script that has been created is called `project_run.tcl`. Before you run the TCL script, the following things need to be checked/changed. First, ensure that you have the correct directory structure and files. For this you should have a directory called 'Vivado' within this directory you should have the file 'project_run.tcl', and the subdirectories 'sources_run' and 'COE_files'. In the 'sources_run' directory you should have all the Verilog files that are used in the project, and in the 'COE_files' directory you should have all COE files that being used in the project. Next, you need to open up the file `project_run.tcl`, and change line 7 to correspond to the directory you are running the TCL file from.

```
7 set PROJECT_DIR C:/Users/Isaac/Desktop/FINAL_DOCUMENTS/Vivado
```

Figure 4.10

The line in the .tcl file that needs to be changed to correspond to your directory.

Finally, you need to find every location where the distributed memory generator is created and replace the current path, as shown below, with a complete direct path on your machine to that file. You should be able to do a find and replace by searching for all occurrences of C:/Users/Isaac/Desktop/FINAL_DOCUMENTS/Vivado/COE_files with the direct path on your machine.

```
87  
88 memory_type {rom} CONFIG.coefficient_file {C:/Users/Isaac/Desktop/FINAL_DOCUMENTS/Vivado/COE_files/weights1_0.coe}]
```

Figure 4.11

An example of the direct path that needs to be changed.

Once you have made these changes, you can run the project using the following steps.

1. Open Vivado Shell. For my version of Vivado, this is called the ‘Vivado 2018.2 Tcl Shell’.
2. cd to the directory containing project_run.tcl. In my case, the command used here is, ‘cd C:/Users/Isaac/Desktop/FINAL_DOCUMENTS/Vivado’
3. To run the TCL script enter the command, ‘source project_run.tcl’.
 - a. When you run this command you should see info running across the TCL command screen.
 - b. This process should take a few minutes, and once it is completed it will launch the GUI version of Vivado.

Once you have run these three steps, you have created the project with all the appropriate source files in Vivado for use on the Kintex Ultrascale xcku035-sfva784-1LV-I part.

4.5.3 Synthesis in Vivado

In the Vivado GUI you can synthesize the entire project by selecting the ‘Synthesis’ button on the left-side of the screen. For the entire project the synthesis takes around 2-3 hours on my machine. The long time required is largely due to the number of IPs that must be synthesized. In order to run the simulation in Vivado with the correct data loaded into the Distributed Block Memories, you must first run the synthesis. For my

project, the synthesis was run for a Kintex-Ultrascale FPGA. This exact FPGA is part xcku035-sfva784-1LV-I in Vivado.

4.5.4 Generating the Waveform in Vivado

In order to check the results of the hardware implementation in Vivado, the analysis of waveforms is used to verify the results. In order to generate the waveform, you first must create a testbench file. The testbench file created here, `tb_digitrec.v`, is stimulating the three inputs to the top module `digitrec.v`. In the testbench file, we are setting the input clock to run at a rate of 100MHz, then we initialize the `en` and `rst` to value of 0. Then after some time, the value of `rst` is set to 1, and then the value of `en` is set to 1. This is because the reset is active low, and must be set to 1 in order for the network to run, and the enable must be set to 1 for the network to run.

Once you have created your test bench, you can add it to the project and start simulation by the following steps.

1. Click 'Add Sources' in the Project Manager on the left of the Vivado GUI.
2. In the 'Add Sources' menu, select 'Add or create simulation sources'.
3. In the 'Add or Create Simulation Sources' menu, click 'Add files', then find the testbench file you created and select 'ok'. Once you have added the testbench file, you can leave all checked values as default and select 'Finish'.
4. Once you have added the testbench file to the project, right-click on 'Simulation > Simulation Settings...' in the 'Flow Navigator' on the left of the Vivado GUI.
5. In the 'Settings' menu that pops up, find 'Simulation top module name:' and change it to 'tb_digitrec', or whatever the name of your top module is.
6. In the 'Settings' menu under the 'Simulation' tab, change 'xsim.simulate.runtime' to 2000ns. In the same tab, make sure 'xsim.simulate.log_all_signals' is checked. You can then press 'OK' in the settings menu.
7. Finally you can go to 'Simulation > Run Simulation > Run Behavioral Simulation' in order to run the simulation.
8. Once the simulation is up, you should see the signals described in the testbench in the waveform viewer. To add all the signals to the waveform viewer, go to the 'Scope' tab and underneath 'tb_digitrec', right-click on 'u_digitrec' and select 'Add to Wave Window'.

After all these steps have been completed, you can see all internal signals of the network.

4.5.5 Interpreting Results in Vivado

The use of the Wave Window in Vivado was important in order to verify that all the entire network is connected correctly, as well as to verify the counters, and the final result of the system.

The first thing that was checked in simulation was the counters. In order to ensure proper functionality of the network, the counters needed to match the design described in Figure 4.7.

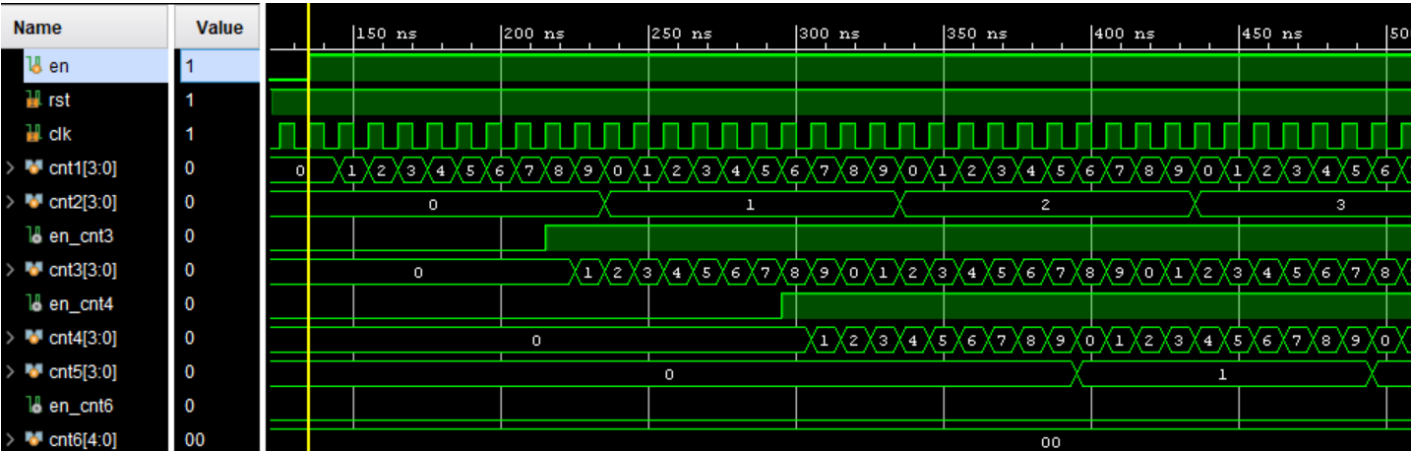


Figure 4.12
Vivado waveform with all counters and their corresponding enable signals.

We can see in the waveform all 6 of the counters, with their appropriate enable signals. This waveform can be compared with the design in Figure 4.7 to see that the first 5 counters are operating as intended. In order to check the operation of the 6th counter, we need to observe its waveform once all the hidden layer neurons have generated their outputs.

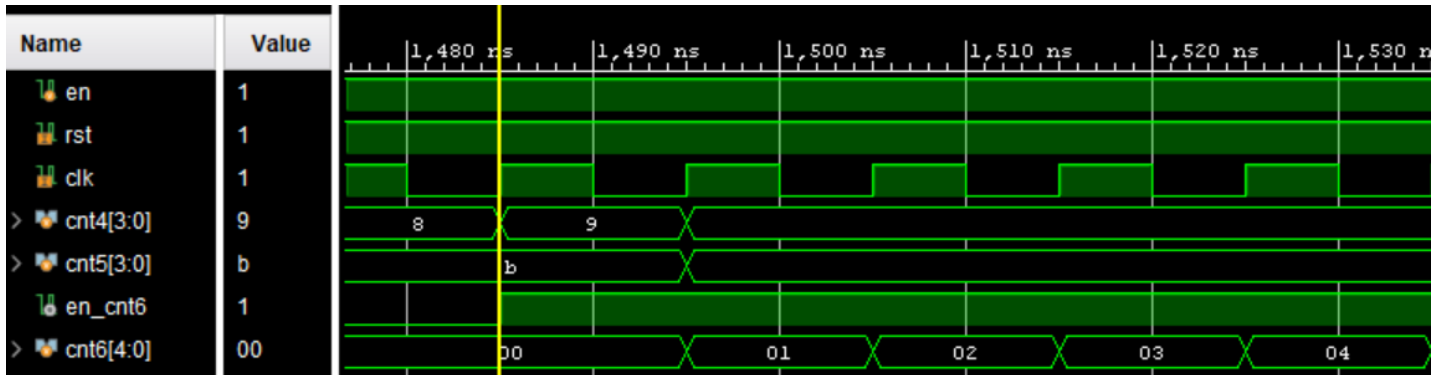


Figure 4.13
Vivado waveform showing when cnt6 starts counting.

By advancing further to when cnt4 = 9, and cnt5 = 0xb or decimal 11, we can see that cnt6 is enabled and begins to count properly.

The final results of the network can be obtained by looking at the value of the wire final_neuron_result_value when cnt6 = 10-19 or 0xa – 0x13. The results are shown on the next page in figure 4.14.

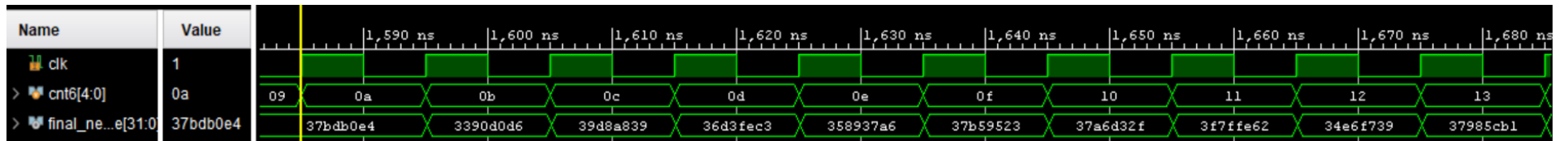


Figure 4.14
Vivado waveform showing the final results of the network.

These results come in such that the value of final_neuron_result_value at cnt6 = 0xa corresponds to detection of a 0, at cnt6 = 0xb corresponds to a detection of a 1, etc. The final detection results of the network shown in the graph below show that the network has strongly detected a '7', with that output being equal to 0.999975324, and all other outputs being nearly 0.

Digit:	Hex Result:	Decimal Result:
0	0x37bd0e4	7.40E-37
1	0x3390d0d6	6.74E-08
2	0x39d8a839	0.000413241
3	0x36d3fec3	6.32E-06
4	0x358937a6	1.02E-06
5	0x37b59523	2.16E-05
6	0x37a6d32f	1.99E-05
7	0x3f7ffe62	0.999975324
8	0x34e6f739	4.30E-07
9	0x37985cb1	1.82E-05

Table 4.3

A table showing the results of the network with an input of a handwritten 7.

When these results are compared with the results obtained in Matlab we can see similar results, with an overwhelming convergence on a value of '7' as the result.

In order to verify that the network was working correctly, other handwritten digit inputs were tested. The output waveform and results for a handwritten 2 are shown below.

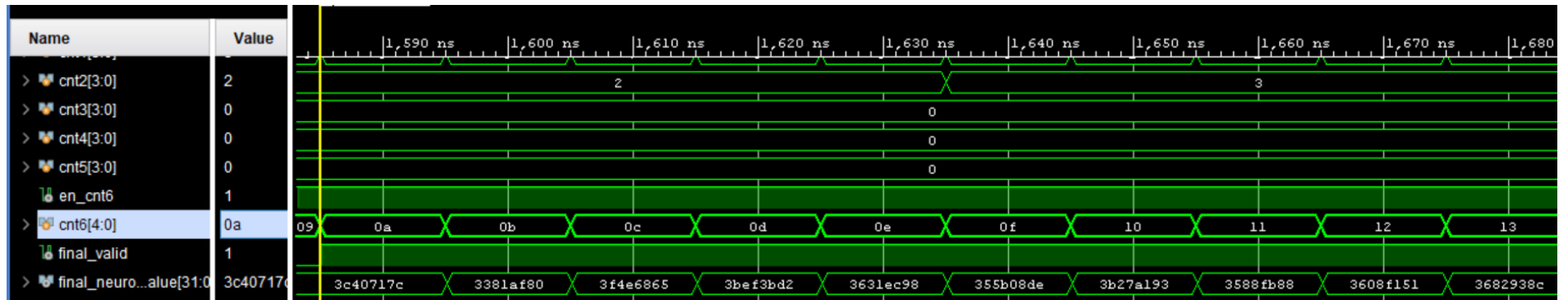


Figure 4.15
Vivado waveform showing the final results of the network for input of 2.

Digit:	Hex Result:	Decimal Result:
0	0x3c40717c	0.011745807
1	0x3381af80	6.04E-08
2	0x3f4e6865	0.80628043
3	0x3bef3bd2	0.007300832
4	0x3631ec98	2.65E-06
5	0x355b08de	8.16E-07
6	0x3b27a193	0.002557848
7	0x3588fb88	1.02E-06
8	0x3608f151	2.04E-06
9	0x3682938c	3.89E-06

Table 4.4

A table showing the results of the network with an input of a handwritten 2.

Again, when these results are compared to the Matlab results, they show the same result, a value of '2' is the detected digit.

The final value tested was with an input of a handwritten 1. The waveform and output are shown in figure 4.16.

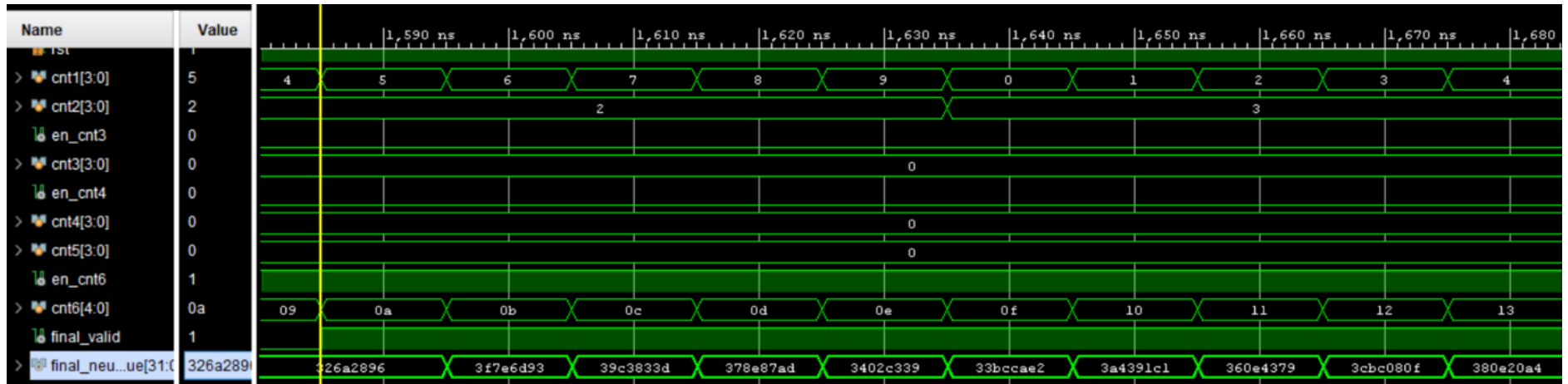


Figure 4.16
Vivado waveform showing the final results of the network for input of 1.

Digit:	Hex Result:	Decimal Result:
0	0x326a2896	1.36E-08
1	0x3f7e6d93	0.99385947
2	0x39c3833d	3.73E-04
3	0x378e87ad	1.70E-05
4	0x3402c339	1.22E-07
5	0x33bccae2	8.79E-08
6	0x3a4391c1	7.46E-04
7	0x360e4379	2.12E-06
8	0x3cbc080f	0.022953061
9	0x380e20a4	3.39E-05

Table 4.5

A table showing the results of the network with an input of a handwritten 1.

These results also show a match with the Matlab results, the final result of the network correctly indicates a detection of a handwritten ‘1’.

In the verification of the results in hardware, only three different input values are considered. The small sample size here is because in order to load the correct input values into the distributed memory generator IP, the hardware must be synthesized. This means that each time we want to change the input image we must re-synthesize the system, which takes another 2-3 hours. Due to this fact 3 correct results which mirror the results seen in Matlab was determined to be enough.

CHAPTER 5:

RESULTS

In this chapter I will discuss the results of the synthesis and simulation of the hardware implementation in Vivado. In this discussion I will compare the results with the software description in Matlab, as well as with related digit detection networks [12,13].

5.1 Execution Time on FPGA

The execution time in for a single digit recognition in hardware can be defined as the time from when the enable for the network goes high, until the last output neuron results is determined. In my implementation running on a 100 MHz clock, this takes 1550ns. This result is compared with the time it takes the network to classify an image in Matlab.

The Matlab execution time is calculated using the ‘tic’ command to begin timing, and ‘time_elapsed = toc’ command to stop timing. This time is taken for processing a total of 10,000 input images. The software time was taken on Matlab on my personal computer that has the following processor, Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz, 2904 Mhz, 2 Core(s), 4 Logical Processor(s).

The execution time in software (Matlab), varies every time you run it, with the first instance being significantly more than each thereafter. This is most likely due to the fact that when the program is run multiple times in a row, it reuses the existing arrays on the Matlab workspace. Because of this, the Matlab code was timed for 10 consecutive runs, with the longest total execution time taken into consideration, and the fastest taken into consideration. The execution time for the software implementation (Matlab) is based upon 10,000 input images, so the total execution time is divided by 10,000 in order to find the execution time per image. The Speedup is found by the formula below.

$$Speedup_A = \frac{(Execution\ Time)_B}{(Execution\ Time)_A}$$

		Total Execution Time (10,000 images) [s]	Execution Time per Image [s]	Speedup	
Software (Matlab)	Fastest	0.6261488	6.26149E-05	0.02475	over Hardware
	Longest	1.9718928	0.000197189	0.00786	over Hardware
Hardware (Verilog)		n/a	0.00000155	40.3967	over Fastest
				127.219	over Longest

Table 5.1

A comparison of the execution times in software and hardware.

From these results, we are able to see that in hardware, the time required to classify an image is 1550 nanoseconds. In software, the longest measured time per image was 1.97E-4 seconds, and the fastest was 6.26E-5 seconds. This led to a speedup in hardware of 127.219 over the longest software run, and a speedup of 40.3967 over the fastest software run.

With these results, the software is also running at 2.7 GHz, which is a 27× higher clock rate than the 100MHz that the hardware implementation results are obtained at. When you take this into consideration, the speedups of the hardware vs the software are both multiplied by 27 to 1090 and 3434 over the fastest and slowest results respectively. These results prove that the implementation of a solution in hardware allows significant speedup over a software implementation.

5.2 Resource Cost on FPGA

As stated in Chapter 4, the synthesis was run for a Kintex-Ultrascale FPGA. This exact FPGA is part xcku035-sfva784-1LV-I in Vivado. A summary of the utilization results of my design is shown below.

Resource	Utilization	Available	Utilization (%)
LUT	44668	203128	21.99
LUTRAM	32	112800	0.03
FF	14274	406256	3.51
DSP	604	1700	35.53
IO	3	468	0.64
BUFG	1	480	0.21

Table 5.2
The utilization results of the hardware synthesis.

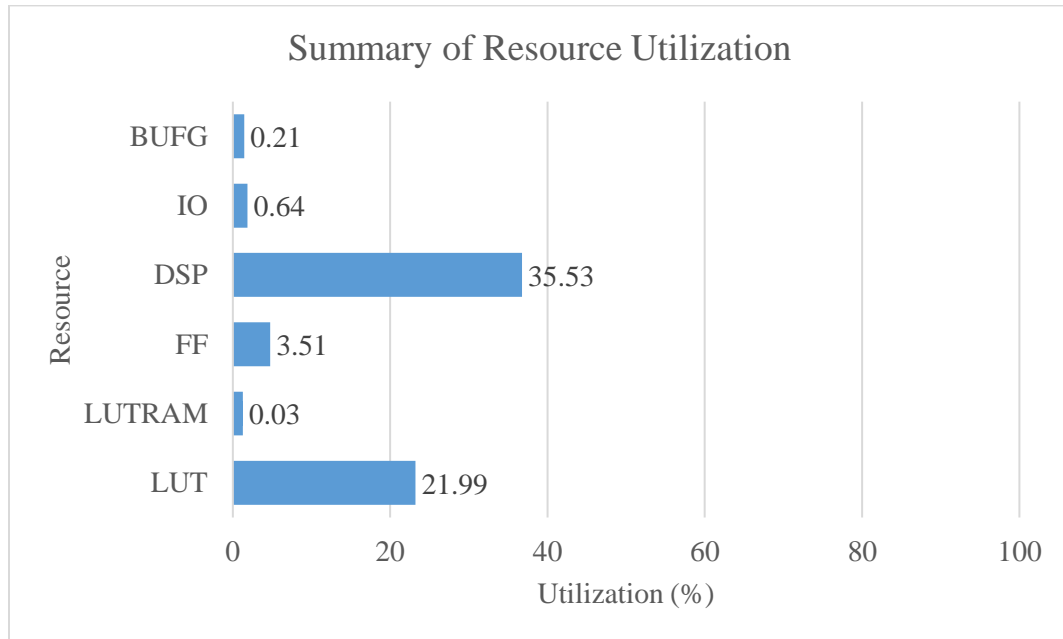


Figure 5.1
A graph of the utilization results.

From the overall summary of utilization results, we can see that the resources needed fit within the bounds of the Kintex-Ultrascale FPGA we are using. From the graph above, we can see that the LUT and DSP are the highest utilized components at 21.99% and 35.53% respectively. These values are as high as they are due to the number of IPs used for floating-point operation, and the specification of ‘Full Usage’ or ‘Max Usage’ of DSPs when they were created.

A further breakdown of the utilization is shown below, with the top module, digitrec, and all the of the submodules found within it.

	CLB LUTs	CLB Registers	DSPs
Total	203128	406256	1700
digitrec (top)	44668	14274	604

acc_add	1880	656	16
mul98	30100	11255	488
mul_12	3473	1330	58
u0_neuron_finish	1575	311	21
u1_neuron_finish	1543	311	21
all blk_mem (total)	6097	411	0

Table 5.3

A further breakdown of the utilization results.

From these results, we can see digitrec contains the total number of CLB LUTs, CLB Registers, and DSPs for the entire system. Below is the data for each instantiated submodule. Finally, the last data point, all blk_mem, represents the totals for all the distributed block memories in the system. These values were combined because there are a total of 210 separate instantiations.

From these results, we can see that most of our resource utilization comes from the mul98 module, which includes 98 single-precision floating point multipliers, and 97 single-precision floating point adders. As I stated above, the utilization of resources is heavily tied to the number of floating point IP that is used. For my applications, this utilization is satisfactory. In future work, if you cut the number of multipliers used in half, you could lower the overall utilization of CLB LUTs, CLB Registers, and DSPs each by around 33%. This will be further discussed in Future Work section of Chapter 6.

5.3 Power Cost on FPGA

The Power Report is generated by Vivado after synthesis has occurred [16,18]. In this report the power is estimated based on the part that is used, and the synthesized netlist. The summary of the power output is shown below.

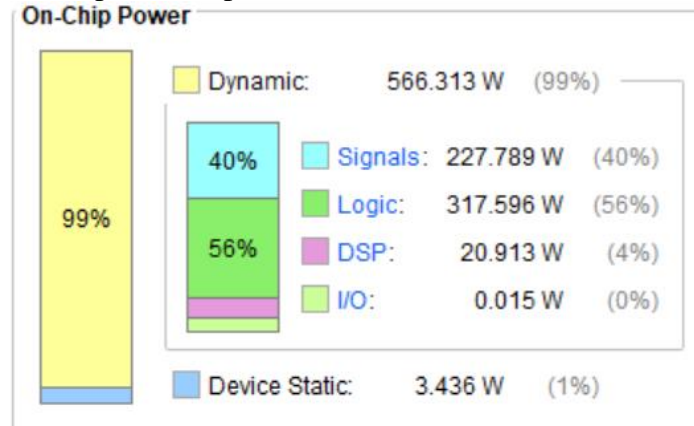


Figure 5.2
An image of the power breakdown.

From the results of the power utilization, the total on-chip power is 569.749 W. Of this power 566.313 W is dynamic power and 3.436 W is static. This gives a breakdown of 99% dynamic power, and 1% static power. A further hierarchical breakdown of the power consumption of the top power modules is shown below.

Utilization	Name	Signals (W)	Data (W)
▼ 566.313 W (99% of total)	digitrec		
> 359.889 W (63% of total)	u0_mul98 (mul_98)	156.188	153.92
> 44.073 W (8% of total)	u0_mul_12 (mul_12)	19.749	19.556
32.657 W (6% of total)	Leaf Cells (476)		
> 19.636 W (3% of total)	u1_neuron_finish (neuron_finish)	5.754	5.684
> 18.422 W (3% of total)	u0_neuron_finish (neuron_finish__xdcDup__1)	5.21	5.153
> 17.023 W (3% of total)	u0_acc_add (acc_add)	6.934	6.934

Figure 5.3
A further breakdown of the power consumption.

In the above image, we can see that the modules with instantiated IP for floating point operations take most of this dynamic power (491.7 W / 566.313 W). The majority of the remaining 74.613 W of dynamic power are related to the distributed block memory that is used to store the input pixels, weights, and biases. The power required for my implementation is fairly high due to the pipelined nature and high utilization of resources. The Vivado floating point IPs are use a lot of power as well, if the number is able to be reduced or a design of our own is created, this could lower the overall dynamic power as well. Further methods for power reduction are discussed in future works.

5.4 Comparison to Related Works

In [12] and [13] the authors have proposed two different network designs for classification of handwritten digits based upon the MNIST dataset. In [12] they describe a 2-layer, 8-bit MLP network and in [13] an 8-bit Super Skinny CNN is used.

5.4.1 Accuracy Comparison

The accuracy comparison is based on the 10,000 test images from the MNIST data set. The results are shown in the following chart.

	Digit Recognition Accuracy (%)
[12] Implementation	89
[13] Implementation	98.8
My Implementation	93.25

Table 5.4
Accuracy comparison to related works.

5.4.2 Speed Comparison

In the implementation of [12], they are running at 25 MHz, and training the network based upon the 55,000 input images, and then classifying the 10,000 test images. I compare the time taken to classify 10,000 images in my implementation and their

implementation by dividing their total time by 4, given that their operating frequency is 25 MHz, and mine is 100 MHz.

In the implementation of [13], they are running at 30 MHz for the parallel channel implementation of SS-CNN and then classifying the 10,000 test images. In order to compare this time, I divide their total time by 10/3 to allow comparison to my 100 MHz frequency.

I then multiply the execution time of classifying 1 image in my network by 10,000 to get the time to classify all the training images using my implementation.

	Total Execution Time (10,000 images) [s]	Speedup	
[12] Implementation	0.95	0.01637	over my Implementation
[13] Implementation	0.66	0.02356	over my Implementation
My Implementation	0.01555	61.0932	over [12]
		42.4437	over [13]

*Table 5.5
Speed comparison to related works.*

The execution time for my system is greatly less than [12] because the weights and biases for my system are trained beforehand, while the total execution time for [12] also includes training based on 55,000 images. The execution time for my system is also greatly less than [13], which also has a greater operational overhead.

These results show that my implementation when compared to related works shows a large speedup. This can be attributed to the fact that the training of the weights and biases is done independently beforehand, and the large utilization of resources used.

5.4.3 Utilization Comparison

A comparison of the utilization of my network with related works is shown below.

	Total Logic Elements
[12] Implementation	34,000
[13] Implementation	98,000
My Implementation	44,668

Table 5.6
A utilization comparison to related works.

From these results, we can see that my utilization is between the two different related works. As stated previously, the high utilization of my network can be attributed to the large number of single-precision IP that I used. If I am able to cut just the number of multipliers in the mul98 module in half, I would be able to lower my logic elements to less than [12].

5.4.4 Summary of Comparison

The results of the comparison of my system with related works shows that all the designs were able to effectively recognize handwritten digits at a high precision, with a significant speedup over pure software implementations. My design converges from these works in that the training of the network was done in software outside of the hardware implementation, with the weight and bias values being imported. This is able to lower the latency and hardware needed to train the network in hardware. The SS-CNN described in [13] is able to recognize digits with near perfect accuracy, but the complexity of the network and design makes the utilization much larger. Overall, my implementation was able to accomplish something in between [12] and [13], it is able to detect handwritten digits with accuracy somewhere between the two, with utilization between the two, but with a significant speedup.

CHAPTER 6:

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The purpose of this paper was to describe the implementation of a robust Multilayer Perceptron (MLP) network for the recognition of handwritten digits. As described, this design can prove as a proof-of-concept for AI-enabled IOT devices.

In choosing my design, I wanted to create as simple of a network as possible that could attain a >92% recognition accuracy with a recognition latency of <3.5ms. In order to attain these numbers an MLP network was used with 784 input neurons, a single hidden layer of 12 neurons, and 10 output neurons. The training of the networks weights and biases based on the MNIST data set was done in Python using stochastic gradient descent. This off-board training is able to vastly decrease the complexity of the overall final network design, and provide a lower latency when classifying images. This network was then implemented in software and hardware, and compared.

My hardware implementation showed a speedup of 40.3967 over the fastest run time in software. This can be adjusted to a speedup of 1090 when taking the 2.7 GHz clock that the software is running at compared to the 100 MHz clock in my implementation. This network was able to attain an accuracy of 93.25% and a latency of 1.44 microseconds. Both of these values are within the goals stated at the beginning of the project.

When my design was compared with related works in [12] and [13], I found a significant speedup over both designs with an accuracy and utilization between the two. These results show that my design represents a middle ground in terms of digit recognition accuracy and utilization that is not achieved by either of these designs.

The power consumption for my design was high, with dynamic power being 99% of the overall consumption at 566 W out of a total of 569 W. Further analysis of this consumption shows that 491.7 W of 566.313 W of dynamic power comes from submodules with instantiated floating-point IPs. In future works, using custom designed floating-point IP could help lessen this power consumption.

Overall, my design has been successful in providing a low-latency, high accuracy design for digit recognition. Due to the proof-of-concept and modular nature of this design, specifics such as the utilization of resources, latency, and overall network design can be changed in order to meet the needs of the end user. Further future modifications and improvements are described in the future work section.

6.2 Future Work

The future work for the proposed network is nearly endless, and can be adjusted based upon the users end-goals for the system, whether it be low power consumption or high digit recognition accuracy. In expanding the system, if floating point operators were created that could be used for multiplication, addition, etc. instead of the IPs the user could likely lower the overall utilization of resources and power consumption. Also, the Vivado IPs limit the lowest precision we can use (single-precision). In software testing, we were able to find that half-precision implementation showed no degradation in the digit recognition accuracy.

Ultimately, this design will be integrated with an image/video processing system including an OV7670 camera and VGA-interface monitor [34,35], in order to have real-time digit recognition. The final edge goal will be a low-cost system-on-chip (SoC) with a reduced complexity on-chip bus architecture proposed in [17,42], which will be a demonstration for time-sensitive applications like cooperative robotics, unmanned aerial

systems, as well as autonomous vehicles. Given the current latency, this is much faster than the frame rate for any regular camera. Due to this fact, the number of resources used in the implementation can be decreased, in order to lower power and utilization, while still having a latency within the frame rate of the camera. The main area of my design that can be lowered is the hidden layer with submodule with 98 multipliers. I have written the equation for the latency of a single hidden layer neuron.

$$\text{Hidden Layer Latency} = \frac{784}{\# \text{ of Multipliers}} + \text{ceil}(\log_2 \# \text{ of Multipliers}) + 7$$

The implementation that I used has 98 multipliers, and thus a latency for a single hidden layer neuron of 22 clock cycles. If I use a design with 49 multipliers changes the latency for a single hidden layer neuron to 29 clock cycles. This change would only add a total of 77 clock cycles to the entire execution time for a single digit recognition, but could divide the number of logic elements used in the mul98 module by half. Even if we lowered the number of multipliers to 1, for this stage, the entire system would only take 9,455 clock cycles to run, which at 100 MHz is only 0.00009455 seconds per recognition.

These are just a few of the things that can be modified in the future to provide a better overall implementation for the end user. Ultimately, they will have to decide what modifications would fit best for their desired implementation.

REFERENCES

1. K. Vaca, A. Gajjar, and X. Yang, "Real-Time Automatic Music Transcription (AMT) with Zync FPGA," IEEE Computer Society Annual Symposium on VLSI (ISVLSI), PP. 378-384, Miami, FL, US, Jan. 13, 2020.
2. K. Vaca, M. Jefferies, and X. Yang, "An Open Real-Time Audio Processing Platform on Zync FPGA," International Symposium on Measurement and Control in Robotics (ISMCR), Accepted, Sept 19-21, 2019.
3. H. He, L. Wu, X. Yang, and Y. Feng, "Synthesize Corpus for Chinese Word Segmentation," The 21st International Conference on Artificial Intelligence (ICAI), Las Vegas, USA, PP. 129-134, July 29 - August 1, 2019.
4. A. Gajjar, X. Yang, L. Wu, H. Koc, I. Unwala, Y. Zhang, and Y. Feng, "An FPGA Synthesis of Face Detection Algorithm using HAAR Classifiers," Intl. Conference on Algorithms, Computing and Systems (ICACS), PP.133-137, July 27-29, Beijing China, 2018.
5. L. Nwosu, H. Wang, J. Lu, I. Unwala, X. Yang, and T. Zhang, "Deep Convolutional Neural Network for Facial Expression Recognition Using Facial Parts," 2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing (DASC), PP. 1318-1321, Orlando, FL, 2017.
6. X. Fu, J. Lu, X. Zhang, X. Yang, and I. Unwala, "Intelligent in-vehicle safety and security monitoring system with face recognition," 2019 IEEE International Conference on Computational Science and Engineering (CSE), PP. 225-229, New York, NY, USA, 2019.
7. Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma, and BeiYub, "Recent advances in convolutional neural network acceleration," Neurocomputing, Vol. 323, No. 5 PP. 37-51, January 2019.

8. S. Gupta, M. Imani, H. Kaur and T. S. Rosing, “NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration,” *IEEE Transactions on Computers*, Vol. 68, No. 9, PP. 1325-1337, Sept. 2019.
9. P. Lei, J. Liang, Z. Guan, J. Wang and T. Zheng, “Acceleration of FPGA Based Convolutional Neural Network for Human Activity Classification Using Millimeter-Wave Radar,” in *IEEE Access*, vol. 7, pp. 88917-88926, 2019.
10. E. Wu, X. Zhang, X. Zhang, D. Berman, I. Cho, and J. Thendean, “Compute-Efficient Neural-Network Acceleration,” *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, PP. 191–200, Feb. 2019.
11. J. Qiao, G. Wang, W. Li, and M. Chen, “An adaptive deep Q-learning strategy for handwritten digit recognition,” *Neural Networks*, Vol. 107, PP. 61-71, November 2018.
12. J. Si and S. L. Harris, “Handwritten digit recognition system on an FPGA,” 2018 *IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, PP. 402-407, 2018.
13. J. Si, E. Yfantis and S. L. Harris, “A SS-CNN on an FPGA for Handwritten Digit Recognition,” 2019 *IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 88-93, 2019.
14. F. Chen, N. Chen, H. Mao, and H. Hu, “Assessing four Neural Networks on Handwritten Digit Recognition Dataset (MNIST),” *Computer Vision and Pattern Recognition*, Nov. 2018.
15. Y. LeCun, C. Cortes, and C. Burges, “MNIST handwritten digit database,” 2010.

16. X. Yang, N. Wu, and J. Andrian, "A Novel Bus Transfer Mode: Block Transfer and A Performance Evaluation Methodology," Elsevier, Integration, the VLSI Journal, Vol. 52, PP. 23-33, Jan. 2016.
17. X. Yang and J. Andrian, "A High Performance On-Chip Bus (MSBUS) Design and Verification," IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (TVLSI), Vol. 23, Issue: 7, PP. 1350-1354, Sept. 2015.
18. X. Yang and J. Andrian, "A Low-Cost and High-Performance Embedded System Architecture and An Evaluation Methodology," IEEE Computer Society Annual Symposium on VLSI (ISVLSI), PP. 240-243, Tampa, FL, USA, Sept. 2014.
19. Y. Zhang, X. Yang, L. Wu, J. Lu, K. Sha, A. Gajjar, and H. He, "Exploring Slice-Energy Saving on An Video Processing FPGA Platform with Approximate Computing," Intl. Conference on Algorithms, Computing and Systems (ICACS), PP.138-143, July 27-29, Beijing China, 2018.
20. Jain S. and Chauhan R. "Recognition of Handwritten Digits Using DNN, CNN, and RNN," Advances in Computing and Data Sciences, Vol 905. PP. 239-248, 2018.
21. Zhan H., Wang Q., Lu Y. "Handwritten Digit String Recognition by Combination of Residual Network and RNN-CTC," Neural Information Processing, Vol. 10639, 2017.
22. C. Gao, D. Neil, E. Ceolini, S. Liu, and Delbruck, "DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator," Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, PP. 21–30, Feb. 2018.
23. FPGA-based Accelerators of Deep Learning Networks for Learning and Classification: A Review <https://arxiv.org/pdf/1901.00121.pdf>

24. Batt, Simon. "The Advantages of Hardware Acceleration in Edge Devices." IoT Tech Trends, 18 Feb. 2019, www.iottechrends.com/advantages-of-hardware-acceleration-edge-devices/.
25. Clabaugh, Caroline, et al. "Neural Networks." Neural Networks, Stanford University, 2000, cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html.
26. Nielsen, Michael. "Neural Networks and Deep Learning." Neural Networks and Deep Learning, Determination Press, 2015, neuralnetworksanddeeplearning.com/chap1.html.
27. Jaspreet. "A Concise History of Neural Networks." Medium, 13 Aug. 2016, towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec.
- 28 A. Gajjar, X. Yang, H. Koc, I. Unwala, L. Wu, and J. Lu, "Mesh-IoT Based System For Large-Scale Environment," 5th Annual Conf. on Computational Science & Computational Intelligence (CSCI), PP. 1019-1023, Las Vegas, NV, USA, 2018.
- 29 A. Gajjar, S. Dave, and X. Yang, "An IoT-Edge-Server System with BLE Mesh Network, LBPH, and Deep Metric Learning," The 22nd Int'l Conf on Artificial Intelligence (ICAI), Accepted, March 2020.
- 30 X. Yang, L. Wu, X. He, A. Gajjar, and Y. Feng, "A Vision of Fog Systems with Integrating FPGAs and BLE Mesh Network," Journal of Communications (JoC) , Vol. 14, No. 3, PP. 210-215, March 2019.
- 31 X. Yang and X. He, "Establishing a BLE Mesh Network using Fabricated CSRmesh Devices," The 2nd ACM/IEEE Symposium on Edge Computing (SEC), Article No. 34, San Jose/Fremont, CA, US, 2017.
- 32 A. Gajjar, Y. Zhang, and X. Yang, "Demo Abstract: A Smart Building System Integrated with An Edge Computing Algorithm and IoT Mesh Networks," The

- Second ACM/IEEE Symposium on Edge Computing (SEC), Article No. 35, San Jose/Fremont, CA, US, 2017.
- 33 H. He, L. Wu, X. Yang, H. Yan, Z. Gao, Y. Feng, and G. Townsend, “Dual Long Short-Term Memory Networks for Sub-Character Representation Learning,” The 15th Intl. Conference on Information Technology - New Generations (ITNG), Las Vegas, NV, USA, 2018.
- 34 Y. Zhang, X. Yang, L. Wu, and J. Andrian, “A Case Study On Approximate FPGA Design With an Open-Source Image Processing Platform,” IEEE Computer Society Annual Symposium on VLSI (ISVLSI), PP. 372-377, Miami, FL, US, Jan. 13, 2020.
35. X. Yang, Y. Zhang, and L. Wu, “A Scalable Image/Video Processing Platform with Open Source Design and Verification Environment,” 20th Intl. Symposium on Quality Electronic Design (ISQED), PP. 110-116, Santa Clara, CA, US, April 2019.
- 36 X. Yang and W. Wen, “Design of A Pre-Scheduled Data Bus (DBUS) for Advanced Encryption Standard (AES) Encrypted System-on-Chips (SoCs),” The 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), PP. 1-6, Chiba, Japan, 2017.
- 37 X. Yang, W. Wen, and M. Fan, “Improving AES Core Performance via An Advanced IBUS Protocol,” ACM Journal on Emerging Technologies in Computing (JETC), Vol. 14, No. 1, PP. 61-63, Jan. 2018.
- 38 M. Fan, Q. Han, and X. Yang, “Energy Minimization for On-Line Real-Time Scheduling with Reliability Awareness,” Elsevier Journal of Systems and Software (JSS) , Vol. 127, PP. 168-176, May 2017.

- 39 X. Yang, N. Wu, and J. Andrian, “Comparative Power Analysis of An Adaptive Bus Encoding Method on The MBUS Structure,” *Journal of VLSI Design*, Vol. 2017, Article ID 4914301, PP. 1-7, May 2017.
40. P. Vangali and X. Yang, “A Compression Algorithm Design and Simulation for Processing Large Volumes of Data from Wireless Sensor Networks,” *Communications on Applied Electronics (CAE)*, Vol. 7, Issue 4, PP. 1-5, July 2017.
- 41 J. Thota, P. Vangali, and X. Yang, “Prototyping An Autonomous Eye-Controlled System (AECS) Using Raspberry-Pi on Wheelchairs,” *Intl. Journal of Compt. Application (IJCA)*, Vol. 158, Issue: 8, PP. 1-7, Jan. 2017.
42. X. Yang and J. Andrian, “An Advanced Bus Architecture for AES-Encrypted High-Performance Embedded Systems,” *US20170302438A1*, Oct. 19, 2017.
43. X. Zhang, J. Lu, X. Fu, X. Yang , I. Unwala, and T. Zhang, “Tracking of Targets in Mobile Robots Based on Camshift algorithm,” *International Symposium on Measurement and Control in Robotics (ISMCR 2019)*, PP. B2-3-1-B2-3-5., UHCL, Houston, USA, 2019.
44. S. Sha, Ajinkya S. Bankar, X. Yang, W. Wen, and G. Quan, “On Fundamental Principles for Thermal-Aware Design on Periodic Real-Time Multi-Core Systems,” *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, Accepted, 2019.
45. W. McCulloch, P. Walter, “A Logical Calculus of Ideas Immanent in Nervous Activity,” *Bulletin of Mathematical Biophysics*, Vol. 5, No. 4, PP. 115–133, 1943.

46. S.C. Kleene, “Representation of Events in Nerve Nets and Finite Automata,” *Annals of Mathematics Studies*, Princeton University Press, PP. 3–41. Retrieved 17, 1956.
47. S. Linnainmaa, “The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors,” *University of Helsinki*, PP. 6–7, 1970.
48. S. Linnainmaa, “Taylor expansion of the accumulated rounding error” *BIT Numerical Mathematics*, Vol. 16, No. 2, PP. 146–160, 1970.
49. C. Mead, M. Ismail, “Analog VLSI Implementation of Neural Systems,” *The Kluwer International Series in Engineering and Computer Science*. 80. Norwell, MA: Kluwer Academic Publishers, ISBN 978-1-4613-1639-8, 1989.
50. A. Ng and J. Dean, “Building High-level Features Using Large Scale Unsupervised Learning,” *arXiv:1112.6209*, 2012.
51. I. Goodfellow, Y. Bengio, and A. Courville, “Deep Learning,” MIT Press, 2016.
52. A. Graves and J. Schmidhuber, “Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks,” *Advances in Neural Information Processing Systems*, Neural Information Processing Systems (NIPS) Foundation, PP. 545–552, 2009.
53. A. Graves, “Novel Connectionist System for Improved Unconstrained Handwriting Recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 31 No. 5, PP. 855–868, 2009.
54. AI Chip Market Report, <https://www.alliedmarketresearch.com/artificial-intelligence-chip-market>, 2019.