

Artificial Neural Network Controller for Single Digit Pattern Recognition using FPGA

by

**Paul Kenneth V. Rigor
Darren B. Soriano
Maria Kym B. Velayo**

A Thesis Report Submitted to the School of Electrical Engineering,
Electronics Engineering, and Computer Engineering
in Partial Fulfillment of the Requirements for the Degree

Bachelor of Science in Computer Engineering

Mapua Institute of Technology

September 2011

APPROVAL SHEET

This is to certify that we have supervised the preparation of and read the practicum paper prepared by **Paul Kenneth V. Rigor, Darren B. Soriano, and Maria Kym B. Velayo** entitled **Programmable Logic Device-Based of an Architecture of an Artificial Neural Network Controller for Pattern Classification Applications** and that the said paper has been submitted for final examination by the Oral Examination Committee.

Mr. Edward Jay M. Quinto
Reader

Analyn N. Yumang
Academe Adviser

As members of the Oral Examination Committee, we certify that we have examined this paper and hereby recommend that it be accepted as fulfillment of the thesis requirement for the Degree **Bachelor of Science in Computer Engineering**.

Engr. Glenn O. Avendaño
Panel Member 1

Engr. Jerry V. Turingan
Panel Member 2

Engr. Dionis A. Padilla
Chairman

This practicum paper is hereby approved and accepted by the School of Electrical Engineering, Electronics Engineering, and Computer Engineering as fulfillment of the thesis requirement for the Degree **Bachelor of Science in Computer Engineering**.

Dr. Felicito S. Caluyo
Dean, School of EECE

ACKNOWLEDGEMENT

First of all, the group is very grateful to the Lord Almighty for granting them the wisdom and knowledge that made this thesis possible, for giving them the strength that keep them standing and for the hope that keep them believing.

The researchers want to express their deepest gratitude to all those people who have given their support in this thesis.

To their families, who gave us not only financial support but also inspiration.

To their adviser, Engr. Analyn N. Yumang, for she has made available her full support in a number of ways; and for her encouragement, guidance and support from the initial to the final stage of the thesis writing.

To Mr. Leonardo P. Nicdao who, for the past few terms assisted and provided the researchers time to use the Microprocessors Laboratory to be able to use the necessary equipment they needed. To the DOIT personnel who assisted them in providing licenses for the Labview. Also, to the CDM personnel and electronics lab assistant who helped them in their PCB.

To their course coordinators, Engr. Noel Linsangan and Engr. Ayra Panganiban, who provided them guidelines in making the thesis and taught them to be more responsible for their own good. To the panels, Engr. Dionis A. Padilla, Engr. Glenn O. Avendaño, Engr. Jerry V. Turingan, Engr. Jocelyn F. Villaverde and Engr. Edzel Geronimo, who scrutinized the thesis and made the researchers be able to improve it.

To their peers who supported them in simple ways and contributed necessary information needed for this thesis. The researchers offer their gratitude and blessings to all those supported from the topic proposal up to the completion of the thesis.

TABLE OF CONTENTS

TITLE PAGE	i
APPROVAL PAGE	ii
ACKNOWLEDGEMENT	iii
TABLE OF CONTENTS	iv
LIST OF TABLES	v
LIST OF FIGURES	vi
ABSTRACT	vii
Chapter 1: INTRODUCTION	1
Chapter 2: REVIEW OF RELATED LITERATURE	4
Artificial Neural Network	5
Network Architecture	8
Perceptron	10
Learning Process	13
Perceptron Learning Rule	14
Back-Propagation Algorithm	14
Related Studies	15
Neural Network Toolbox	19
Field Programmable Gate Array	20
Chapter 3: ARTIFICIAL NEURAL NETWORK CONTROLLER FOR SINGLE DIGIT PATTERN RECOGNITION USING FPGA	22
Abstract	22

Introduction	22
Methodology	25
Results and Discussion	32
Conclusion	59
References	50
Chapter 4: CONCLUSION	52
Chapter 5: RECOMMENDATION	54
REFERENCES	55
APPENDICES	57
Appendix A	58
Appendix B	75

LIST OF TABLES

Table 2.1:	Recognition rate per digit comparison table for MLP(SVM), MLP and SLP	18
Table 2.2:	Multiple SVM Classifier vs. MLP Classifier	18
Table 3.1:	Training data patterns.	28
Table 3.2:	Test data patterns.	29
Table 3.3:	Performance and complexity comparison table for SLP and MLP.	32
Table 3.4:	MLP Pruning table.	33
Table 3.5:	SLP accuracy measurement table for software simulation.	38
Table 3.6:	Sample MLP accuracy measurement table for software simulation.	43
Table 3.7:	SLP accuracy measurement table for hardware simulation.	48
Table 3.8:	Performance comparison table for the hardware and software simulation.	48

LIST OF FIGURES

Figure 2.1:	Simplified configuration of an organic neuron.	6
Figure 2.2:	Artificial model of a neuron.	8
Figure 2.3:	Example of a single-layer feedforward network.	9
Figure 2.4:	Example of a multilayer feedforward network.	10
Figure 2.5:	Example of linearly separable patterns.	10
Figure 2.6:	An Exclusive-OR function.	11
Figure 2.7:	Architectural graph of a multilayer perceptron with two hidden layers.	12
Figure 2.8:	Simple memory model.	16
Figure 2.9:	Architecture of the hybrid network.	17
Figure 2.10:	Sample result of a neural network trained to recognize some polygons.	16
Figure 2.11:	LabVIEW FPGA block diagram.	21
Figure 3.1:	Flowchart of the incremental pruning algorithm.	27
Figure 3.2:	Neural network model selection decision tree.	31

ABSTRACT

The purpose of this study is to design an artificial neural network for single digit pattern recognition, making use of a field programmable gate array (FPGA) as the main component in implementing its architecture. The study aims to determine which of the two neural network models, single layer perceptron model (SLP) with perceptron learning rule and multilayer perceptron (MLP) model with backpropagation learning, will be used in the implementation of the architecture of the artificial neural network controller. The software simulation shows that the SLP results to 94.13% mean accuracy while the MLP results to 94.96% mean accuracy. With only 0.8779% difference between the mean accuracies of both model, the complexity of each model is considered as the main criterion in selecting the model to be implemented in hardware. With a total number of 25 neurons in its structure, the SLP is chosen for the hardware implementation. The hardware implementation gives 95.60% mean accuracy. The percentage difference between the results in software simulation and hardware simulation is 1.5496%. The result of the hardware implementation is consistent with the result of the software simulation. The outcome of both simulations suggests that the neural network model implemented in hardware is accurate and effective in recognizing patterns presented to it. The system is able to learn and recognize perfect and corrupted single digit patterns accurately.

Keywords: artificial neural network, FPGA, perceptron, neurons, backpropagation

CHAPTER 1

INTRODUCTION

Artificial neural networks are simple abstractions of the extremely complex network of biological neurons found in humans and animals, which could be realized as elements in a program or as circuits made of silicon. They consist of a large number of highly interconnected processing elements working together in parallel to solve problems. Like humans, they learn from example; albeit they do not possess a fraction of the power of the human brain, they can be trained to perform useful functions. Such capability made them useful in providing solutions to paradigms that linear systems have difficulty or are not able to solve, like adapting to circumstances or dealing with noisy environment data. Throughout the years, artificial neural networks reached applications in the design and implementation of intelligent systems, which became a key factor for the innovation and development of better products for society.

At least two phases are necessary for the advancement of a technology: conceptualization and implementation. From the concept of biological neurons, countless researches have been made which realized the defining characteristics and the potential of artificial neurons. At present, they are in the stage where practical applications are appearing. Although artificial neural networks provide elegant solutions to problems which linear systems are unable to resolve, their inherent limitation is that computational resources required to perform the data manipulation are large and complex. This is due to their many non-linear processing elements operating in parallel. Given the processing capabilities of computers nowadays, neural networks could be implemented in software; however, a

hardware to perform neural network computations is very difficult and very costly to implement.

Thus, this thesis was conceptualized because of the need for a practical hardware implementation of a neural network. Initially, why implement neural networks in hardware if one could easily simulate it in software? The main reason is that the hardware implementation of artificial neural networks can take advantage of their inherent parallelism and run orders of magnitude faster than software simulations. There actually exists commercial neural network hardware like neural network processors and neurocomputers, but they lack popularity for they are hard to find, very expensive, and very complex. From this, the main problem of this study was based; that there is no available neural network hardware that is simple yet effective in pattern recognition applications.

The main objective of this study is to design an artificial neural network controller mainly for pattern recognition applications, utilizing a field programmable gate array (FPGA) as the core component in implementing its architecture. The specific objectives of this study are to determine the accuracy and number of neurons used for the single layer perceptron neural network and the multilayer perceptron neural network; to determine if there are significant differences between the accuracies of the said neural network models; to select between the two neural network models that will be more effective to implement in hardware; to determine the accuracy of the selected neural network model in hardware; and to find out if the accuracy of the selected neural network model is consistent with the accuracy in its software implementation.

This study is significant because it simplifies the integration of neural networks to the design of systems involving pattern recognition applications. It takes advantage of the

fact that neural network hardware implementations process data much faster than the neural network software implementations. The proposed artificial neural network controller is most useful to real-time pattern recognition systems such as aircraft component fault detectors, medical diagnosis systems, speech synthesizers, manufacturing process control systems, truck brake diagnosis systems, etc.

The function of the proposed artificial neural network controller shall only focus on single-digit pattern recognition. It shall have two modes of operation: training and operation mode. It will not be capable of learning while in operation mode as learning is only possible while the neural network in training mode. Only a single neural network model with a specific learning algorithm shall be implemented in hardware. Of the numerous neural network models which are available, only two models shall be evaluated and compared: the single layer perceptron (SLP) model and the multilayer perceptron model (MLP). Only a single, specific learning algorithm shall be chosen for each of the neural network models: perceptron learning rule algorithm for SLP and backpropagation learning algorithm for MLP. The learning method that will be used is supervised learning (i.e., an input is presented to the neural network and a corresponding target response is set at the output). The number of inputs and outputs to each of the neural network models shall be restricted to 25 inputs and 25 outputs.

Chapter 2

REVIEW OF RELATED LITERATURE

Many scientists consider the human brain as the most complex object in the universe due to its great structure and capability to come up with its own rules through what we usually refer to as "experience". It has been said that experience is built up over time, with the most dramatic development of the human brain happening during the first two years from birth; but the development continues well beyond that stage (Haykin, 1999). The way humans have acquired the ability to distinguish information can only be through learning experience. By trial and error, they have learned to do certain tasks. Contrary to how humans deal with differing information, machines do not learn. They are preprogrammed to perform certain jobs sequentially or step by step. They are not capable of doing trial and error calculations so whenever problems are encountered during execution, the only way to reduce errors, if not totally eliminate, is to manually reprogram the machine. These machines are usually composed of a large number of complex building blocks; if one unit fails, the whole system malfunctions. The human brain, however, is composed of almost identical building blocks: the neurons, which are electrically excitable cells that transmit and process information through electrical and chemical signaling. Several units may be destroyed without notably affecting the behavior of the whole system. The desire and need to come up with a computer that emulates the behavior of the human brain gave rise to the study and development of artificial neural networks.

Artificial Neural Network

Artificial neural network (ANN) is a computational model that is inspired by the way biological nervous systems, such as the brain, organize and process information. An ANN emulates the capability of a brain to adapt to changes in the surrounding environment and process information in parallel. Neural networks, with their exceptional ability to acquire meaning from complicated or imprecise data, can be used to extract patterns and distinguish trends that may be too complex to be perceived by either humans or other common computing methods. These networks can perform tasks that a linear program cannot. When an element of a neural network fails, it can continue without any problem because of its parallel nature. Unlike machines or conventional computers, neural networks cannot be reprogrammed to improve in doing particular tasks, but instead they are submitted to series of training or learning process. The technique used in this process is commonly referred to as the learning algorithm.

Neural networks and conventional computers are not in competition; instead, they complement each other. There are certain tasks that are more suited to each approach of processing information. Even more, bigger and complex tasks may require the use of both to improve performance and the reliability of the results.

The building unit of a neural network is a simplified model of an organic neuron. An organic neuron, as shown in Figure 2.1, has three parts: a set of incoming fibers (dendrites), a cell body (soma) and an outgoing fiber (axon). It has been said that a neuron can connect and pass its signal to many other neurons (Graupe, 2007). Similarly, since there are many dendrites per each neuron, a single neuron can receive messages from many other neurons. In this manner, the biological neural network interconnects. Neurons generate

electrical pulses and transmit them along the axons to bulb-like structures (synapses) where fibers contact. Electrical activities transferred by the synapse to another neuron may contribute to the excitation or inhibition of that neuron. Veelenturf (1995) stated that synapses play an important role because their transmission efficiency for electrical pulses from an axon to the dendrites (or somas) of other neurons can be changed based on the ‘profitability’ of that alteration. The learning capability of human beings is associated in the facility of altering the transmission efficiency of those synapses. The change of the synaptic transmission efficiency serves as a memory for past experiences.

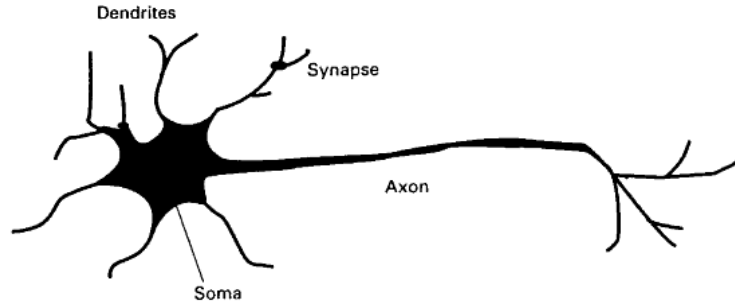


Figure 2.1. Simplified configuration of an organic neuron.

Haykin (1999) stated that an artificial model of a neuron, as shown in Figure 2.2, has three basic elements: a set of synapses or connecting links, an adder for summing the input signals, and an activation function for limiting the amplitude of the output of a neuron.

The synaptic transmission efficiency is represented by a real number w_{kj} , by which a signal x_j at the input of synapse j is multiplied before entering the neuron cell. The number w_{kj} is termed as the synaptic weight input of x_j . Haykin (1999) added that it is important to take note of the manner in which the subscripts of the synaptic weight w_{kj} are written. The first subscript refers to the neuron in question and the second subscript refers to the input end

of the synapse to which the weight refers. The variable x_j , which models the absence or presence of a train of electrical pulses in a real neural fiber, may have the value zero or one. In that case, the network is said to be a binary artificial neuron representing the ‘one-or-zero’ behavior of a real neuron. The artificial model of neuron also includes an externally applied bias b_k . This bias has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative.

All inputs x_j are weighted by the synaptic transmission efficiency and are summed up to one number $u_k = \sum_{j=1}^m w_{kj}x_j$. This linear combiner output u_k ascertains in some way or another the output value y_k of the artificial neuron. The output y_k is computed using the formula $y_k = \varphi(u_k + b_k)$, where φ is the activation function. One commonly used activation function is the Threshold Function. Using this activation function, the output y_k will be one, which indicates that the neuron is firing, if the weighted input exceeds some threshold, and will be zero, indicating that the neuron is silent, if the weighted input is below the threshold. The benefits of an artificial network are mainly the results of the modifiability of behavior by changing the weights w_{kj} in a learning stage.

In an ANN, learning is done by changing the synaptic weights of the network in a methodical fashion to attain a desired design objective.

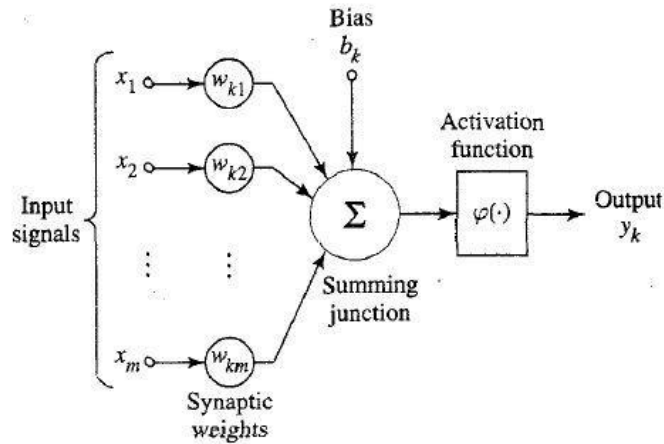


Figure 2.2. Artificial model of a neuron.

Network Architecture

The manner in which the neurons of a neural network are built is closely connected to the learning algorithm used to train the network. Haykin (1999) stated that, in general, there are three fundamentally different classes of network architectures: single-layer feedforward networks, multilayer feedforward networks and the recurrent networks.

The neurons are organized in the form of layers in a layered neural network. In the simplest form of a layered network, there is an input layer of source nodes that projects onto an output layer of neurons, but not vice versa. In other words, this network is feedforward type. In a single-layer feedforward network (Figure 2.3), there is only one output layer of neurons (computation nodes). While in a multilayer feedforward network (Figure 2.4), one or more hidden layers are present, whose computation nodes are correspondingly called hidden neurons or hidden units. In a multilayer network, the output signals of the first layer are used as the inputs to the second layer. Then, the output signals of the second layer are used as the inputs to the third layer, and so on for the rest of the network. Haykin (1999)

stated that the function of hidden neuron is to intervene between the external input and the output in some useful manner. By adding one or more hidden layers, the network is enabled to extract higher-order statistics.

A recurrent neural network distinguishes itself from a feedforward neural network in that it has at least one feedback loop. For example, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons. A recurrent network may have or not have hidden layers or neurons.

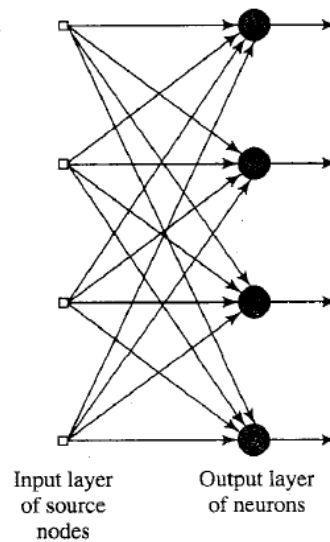


Figure 2.3. Example of a single-layer feedforward network.

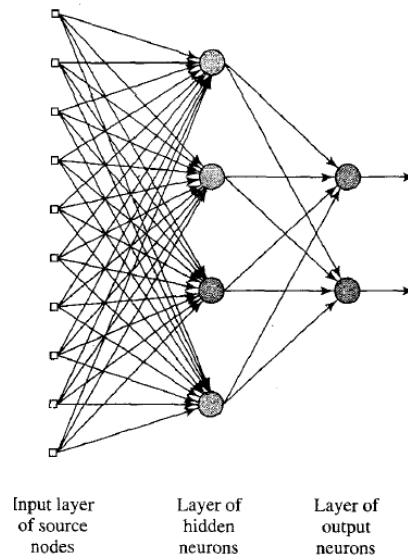


Figure 2.4. Example of a multilayer feedforward network.

Perceptron

Perceptron is a neuron-like artificial learning device which was introduced by Frank Rosenblatt in 1958. It is the simplest form of a neural network which is mainly used for recognition and classification of patterns that are considered linearly separable (Haykin, 1999). Patterns are said to be linearly separable if they can be completely separated by a single line. In general, two sets of pattern are linearly separable if they can be separated by a hyperplane. An example of linearly separable patterns is shown in Figure 2.5.

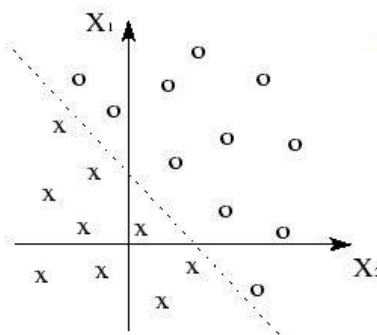


Figure 2.5. Example of linearly separable patterns.

The journal entitled “Introduction to artificial neural networks”, from the Electronics Technology Directions to the Year 2000 (1995), stated that typical artificial neural networks consist of one input layer and one output layer. Such ANNs are commonly referred to as single-layer perceptrons (SLP).

For n binary input variables, there are 2^n possible input patterns, and for 2^n input patterns; there are 2^{2^n} possible logic functions connecting n inputs to one binary output. In the journal, it is stated that a single-layer perceptron is only capable of realizing a small subset of these functions known as linearly separable logic functions or threshold functions. Single-layer perceptrons cannot be used in cases such that of Exclusive-OR function which is not linearly separable. Figure 2.6 shows a truth table pertaining to a two variable Exclusive-OR function. This figure, where four possible combinations of input can be derived, shows that no single straight line can separate the output responses.

Single-layer perceptrons are said to be easy to train and set up. But they can only represent a limited set of functions and perfectly separate linearly separable data.

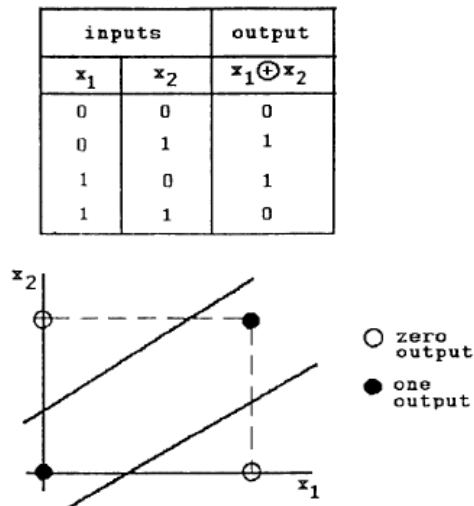


Figure 2.6. An Exclusive-OR function.

Non-linearly separable functions can only be implemented by using multilayer feedforward networks. Typically, these networks consist of an input layer, one or more hidden layers of computation nodes, and an output layer of computation node. The input signal propagates through the network in a forward direction, on a layer-by-layer basis (Haykin, 1999). These neural networks are commonly referred to as multilayer perceptrons (MLP). Multilayer perceptrons have been used successfully to solve some complex problems by training them in a supervised manner with a highly popular algorithm called the error backpropagation algorithm.

A multilayer perceptron has three distinctive characteristics: the model of each neuron in the network includes a nonlinear activation function such as the sigmoidal nonlinearity; the network has additional layers hidden between the input layer and the output layer that enable the network to learn complex tasks by extracting progressively more meaningful features from the input pattern; and the network exhibits a high degree of connectivity determined by the synapses of the network. Figure 2.7 shows an example of a multilayer perceptron that has two hidden layers and an output layer.

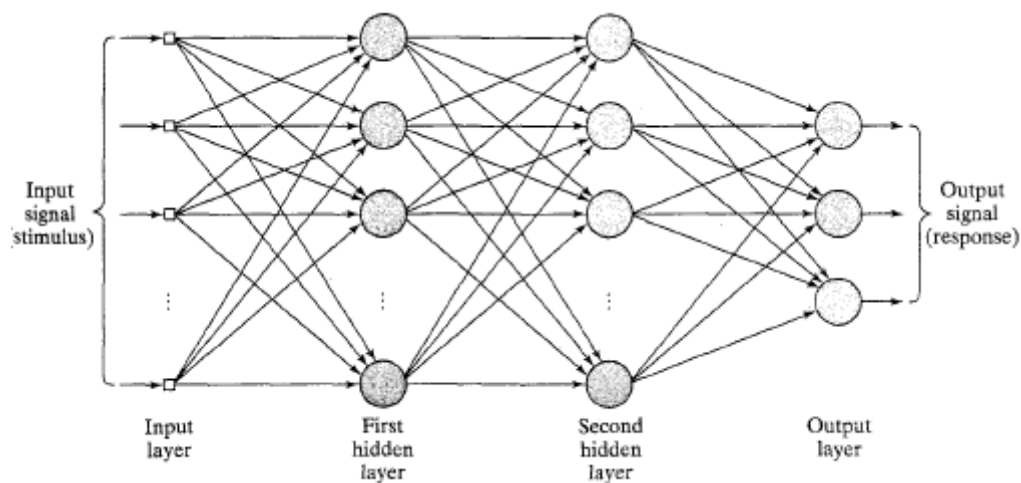


Figure 2.7. Architectural graph of a multilayer perceptron with two hidden layers.

Learning Process

A neural network adapts to its environment through a continuous process of adjusting its synaptic weights and bias levels. It becomes more knowledgeable about its environment after every cycle of the learning process (Haykin, 1999). The rule or technique used in this process is usually termed as the learning algorithm. A learning algorithm is a set of well-defined rules for the solution of a learning problem. Basically, learning algorithms differ from each other in the way parameters, such as synaptic weights, are adjusted in the network. The model of the environment in which the neural network operates is referred to as the learning paradigm.

Two learning paradigms are used in artificial neural networks: learning with a teacher and learning without a teacher. In learning with a teacher, or supervised learning, the knowledge of the environment available to the teacher is transferred to the neural network through training as fully as possible. When this condition is reached, the teacher may be dispensed allowing the neural network deal with the environment completely by itself. In learning without a teacher, as the name implies, there is no teacher to oversee the learning process. There are no labeled examples of the function to be learned by the network. This learning paradigm is subdivided into two: reinforcement learning and unsupervised learning. In reinforcement learning, the learning of an input-output mapping is done through continued interaction with the environment in order to minimize a scalar index of performance. While in an unsupervised learning, once the network has become tuned to the statistical regularities of the input data, it develops the capability to create internal representations for encoding features of the input and thereby create new classes automatically (Haykin, 1999).

The sequence of a learning process starts when the neural network is stimulated by the environment. Next, the network undergoes changes in its parameters as a result of the stimulation. Then, the network responds in a new way due to the changes done. This sequence is repeated until the desired outcome has been attained.

There are three widely used learning rules. These are the Perceptron learning rule, backpropagation algorithm and the support vector machines.

Perceptron Learning Rule

During training, an input is fed into the network and flows through the network generating a set of values on the output units. Then the actual output is compared with the desired target, and a match is computed. If the output and target match, no change is made to the net. However, if the output differs from the target a change must be made to some of the connections. The Perceptron learning rule was introduced by Rosenblatt. It is a typical error connection learning algorithm of single-layer feed-forward networks with linear threshold activation function. It specifies that the change in the weight from one unit to another reflects three factors; first is the learning rate which controls how weights are altered, the error and the input (Fuller, 2010).

Backpropagation Algorithm

Another commonly used rule in a learning process is the backpropagation (BP) algorithm that was based on the basic error-correction learning. It was proposed in 1986 by Rumelhart, Hinton and Williams for setting weights and hence for the training of multilayer perceptrons (Graupe, 2007). The backpropagation algorithm teaches a feedforward

multilayer neural network a given set of input patterns with known classifications. The network analyzes its output response to the sample input pattern when each entry of the sample set is introduced to the network. The output response is then examined in contrast to the known and desired output and then the error value is calculated. Based on the error, the synaptic weights are adjusted. The set of these sample patterns is iteratively presented to the network until the error value is reduced.

Related Studies

Hardware implementation of neural networks may not be that easy because of the complexity of the topic itself, but this cannot be a hindrance in pursuing researches and development projects relating to neural networks. According to F. Kampf, P. Koch, K. Roy, M. Sullivan, Z. Delalic and S. DasGupta (1989), a neural-processing chip was developed to aid the major problem in hardware implementation of neural networks.

H. Faiedh, Z. Gafsi, K. Torki and K. Besbes (2004) proposed in their paper a fully-digital hardware implementation of neural networks where a circuit can recognize a classification task and uses two layers for the solution. The researchers pointed out that a generic structure of a neuron can be used to build a neural network.

M. Badgero (1994) introduced a digital neuron model which he called simple memory model. Its purposes include simplifying the design of digital neural networks and allowing the design of custom hardware using common digital parts. This simple memory model, as shown in Figure 2.8, will need multiple passes in order to solve a given problem and will use a feedback network design to be able to store time-varying patterns. This model

is only used in supervised learning methods which are easy to digitize, such as the XOR function and the binary counter that uses an RS latch.

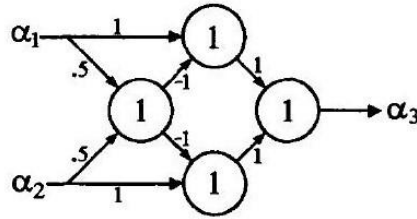


Figure 2.8. Simple memory model.

Neural network can be used in a lot of applications. V. Turchenko, V. Kochan, A. Sachenko, and Th.Laopoulos (2001) made use of neural network, which, according to them, is the most effective way to improve the accuracy of a physical quantity measurement by a sensor. The research concluded confirming the efficiency of using the proposed advance method of historical data integration together with the approximation and prediction of neural networks in error reduction of physical quantity measurement.

H. Raju, R. Hobson and P. Wetzal (2001) proposed in their research an application of a neural network using the architecture of the hybrid network. This architecture of the hybrid network, as shown in Figure 2.9, was used for the visual search classification particularly in classification of geometrical shapes. The network was trained to recognize inputs of shapes with 3, 4, 5, 6, 7, 10, 15, and 20 sides. It took 10 hours for the network's training to be able to recognize the polygon's correct classes. Referring to Figure 2.10, the shape on the left is the actual input pattern and the shape on the right is the answer of the network. It answered decagon for two reasons; first, the network was not trained to recognize a polygon with 8 sides. Second, both input vectors were far apart from any of the

classes. According to the research, this misclassification can be avoided by introducing another class with 8 sides to the network.

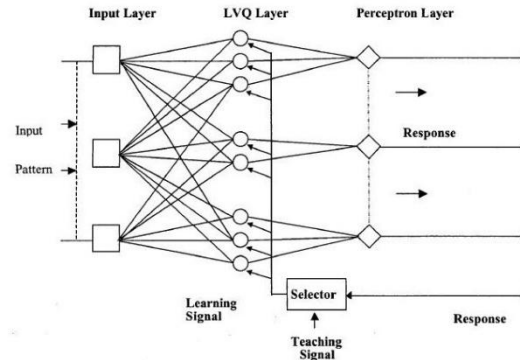


Figure 2.9. Architecture of the hybrid network.

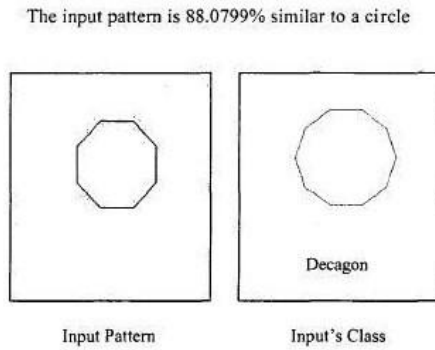


Figure 2.10. Sample result of a neural network trained to recognize some polygons.

A new method for pattern recognition was proposed by A. Pourmohammad and S. Ahadi (2009), which is based on support vector machines (SVMs) for pattern recognition. The research was applied to isolated handwritten Persian digits. The result after simulating the trained network is as follows.

Recognition Rate (%) per digit	MLP(SVM) Train Test	MLP Train Test	SLP Train Test
0	99.6 93	100 94	100 98.8
1	98.1 91	98.3 98	100 97.6
2	98.5 80	96.1 79	100 96.6
3	99.3 87	98.6 85	100 96.4
4	98.1 87	98.5 95	100 98.6
5	98.6 90	98.5 95	100 99.3
6	96.8 69	98.3 79	100 97.9
7	99.5 93	100 99	100 99.7
8	99.5 94	99.6 98	100 99.7
9	98.5 87	98.6 90	100 98.5
Average	98.6 87.1	98.6 91.2	100 98.3

Table 2.1. Recognition rate per digit comparison table for MLP(SVM), MLP and SLP.

Table 2.1 shows a comparison of results between using a multi-layer perceptron (support vector machines), multi-layer perceptron and single-layer perceptron. The result of MLP(SVM) gained the lowest accuracy. This is because the researchers did not include enough sample figures of Persian digits. But if it will be compared with another set of results that made use of enough sample figures for the MLP(SVM), as shown in the table below, the use of MLP(SVM) shows higher recognition rate.

Classifier	Multiple SVM Classifier	MLP Classifier
Recognition Rate (%)	94.14	91.25
Error Rate (%) (Substitution Rate)	5.86	8.75
Rejection Rate (%)	0	0

Table 2.2. Multiple SVM Classifier vs. MLP Classifier

Based on the two results, it only proves that proper training of the network should be done in order to achieve better recognition. Neural networks are not only limited in basic pattern recognition and classification applications. They are also implemented in broad areas of knowledge, particularly in the field of Biometrics. H. Gulcur and G. Buyukaksoy (1992) proposed a method that is based on artificial neural networks for the identification of white corpuscle images. A white corpuscle is a blood cell and is an important part of the body's defense system. The researchers made use of Kohonen's Self-Organizing Map algorithm to be able to compute for the weight vectors that corresponds to the different white corpuscles. The network was able to respond to six different white corpuscles after some time of training. Better result accuracy will demand for a higher picture resolution of the digitize image. The researchers also pointed out that a low resolution was caused by longer time of training. Although this application may seem to be difficult, it is something worth improving because it is an advancement of medical techniques using new technology.

Neural Network Toolbox

Neural networks have been trained to do complex functions in various fields, including pattern recognition, identification, classification, speech, vision, and control systems. These networks can also be trained to tackle problems that are difficult for conventional computers or human beings. The toolbox highlights the use of neural network paradigms that build up to—or are themselves used in—engineering, financial, and other practical applications. There are four ways one can use the Neural Network Toolbox software. The first way is through the four graphical user interfaces (GUIs). The second way to use the toolbox is through basic command-line operations. The command-line

operations offer more flexibility than the GUIs, but with some added complexity. The third is through customization. This advanced capability allows users to create their own custom neural networks, while still having access to the full functionality of the toolbox. The last is through the ability to modify any of the functions contained in the toolbox.

Several tools are available in this software. This study focuses only on using the neural network pattern recognition tool (The MathWorks, Inc., 2010).

Field-programmable Gate Array

Field-programmable gate arrays (FPGAs) are said to be a member of a class of devices called field-programmable logic (FPL). FPLs are identified as programmable devices containing repeated fields of small logic blocks and elements (Meyer-Baese, 2007). According to Meyer-Baese (2007) it can be argued that an FPGA is an ASIC technology since FPGAs are application-specific ICs. It is, however, generally assumed that the design of a classic ASIC required additional semiconductor processing steps beyond those required for an FPL. The additional steps provide higher-order ASICs with their performance and power consumption advantage, but also with high nonrecurring engineering costs. Gate arrays, on the other hand, typically consist of a “sea of NAND gates”, whose functions are user provided in a “wire list.” The wire list is used during the fabrication process to achieve the distinct definition of the final metal layer. The designer of a programmable gate array solution, however, has full control over the actual design implementation without the need and delay for any physical IC fabrication facility (Meyer-Baese, 2007).

The National Instruments LabVIEW FPGA module allows users to develop virtual instruments (VI) that can define custom I/O and control hardware without prior knowledge

of digital design or complex EDA tools. It can be used to prototype and create FPGA code. The block diagram of a LabVIEW FPGA VI can represent the parallelism and timing embedded systems. The LabVIEW FPGA are ideal for applications requiring custom signals, such as integrating custom timing, triggering, and synchronization, parallel processing and data acquisition, custom motion control and off-loading signal processing and control from a host PC or real-time system.

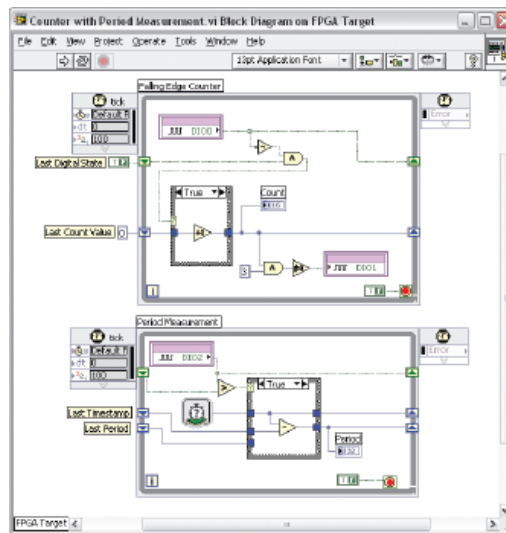


Figure 2.11. LabVIEW FPGA block diagram.

The ideal way to represent the parallelism of FPGA hardware is LabVIEW graphical programming. The compiled LabVIEW FPGA code is implemented in hardware by configuring logic cell resources on the FPGA. The independent sections of code like parallel loops are implemented as independent sections of the FPGA. When the chip is configured, the data is clocked through the device at a rate specified by the onboard clock, executing independent areas of the chip simultaneously. Figure 2.11 illustrates how the parallelism of LabVIEW FPGA enables the loops to execute simultaneously without competing for execution resources (National Instruments, Inc., 2005).

Chapter 3

ARTIFICIAL NEURAL NETWORK CONTROLLER FOR SINGLE DIGIT PATTERN RECOGNITION USING FPGA

Abstract

The purpose of this study is to design an artificial neural network for single digit pattern recognition, making use of a field programmable gate array (FPGA) as the main component in implementing its architecture. The study aims to determine which of the two neural network models, single layer perceptron model (SLP) with perceptron learning rule and multilayer perceptron (MLP) model with backpropagation learning, will be used in the implementation of the architecture of the artificial neural network controller. The software simulation shows that the SLP results to 94.13% mean accuracy while the MLP results to 94.96% mean accuracy. With only 0.8779% difference between the mean accuracies of both model, the complexity of each model is considered as the main criterion in selecting the model to be implemented in hardware. With a total number of 25 neurons in its structure, the SLP is chosen for the hardware implementation. The hardware implementation gives 95.60% mean accuracy. The percentage difference between the results in software simulation and hardware simulation is 1.5496%. The result of the hardware implementation is consistent with the result of the software simulation. The outcome of both simulations suggests that the neural network model implemented in hardware is accurate and effective in recognizing patterns presented to it. The system is able to learn and recognize perfect and corrupted single digit patterns accurately.

Keywords: artificial neural network, FPGA, perceptron, neurons, backpropagation

Introduction

Artificial neural networks are simple abstractions of the extremely complex network of biological neurons found in humans and animals, which could be realized as elements in a program or as circuits made of silicon. They consist of a large number of highly interconnected processing elements working together in parallel to solve problems. Like humans, they learn from example; albeit they do not possess a fraction of the power of the

human brain, they can be trained to perform useful functions. Such capability made them useful in providing solutions to paradigms that linear systems have difficulty or are not able to solve, like adapting to circumstances or dealing with noisy environment data. Throughout the years, artificial neural networks reached applications in the design and implementation of intelligent systems, which became a key factor for the innovation and development of better products for society.

At least two phases are necessary for the advancement of a technology: conceptualization and implementation. From the concept of biological neurons, countless researches have been made which realized the defining characteristics and the potential of artificial neurons. At present, they are in the stage where practical applications are appearing. Although artificial neural networks provide elegant solutions to problems which linear systems are unable to resolve, their inherent limitation is that computational resources required to perform the data manipulation are large and complex. This is due to their many non-linear processing elements operating in parallel. Given the processing capabilities of computers nowadays, neural networks could be implemented in software; however, a hardware to perform neural network computations is very difficult and very costly to implement.

Thus, this thesis was conceptualized because of the need for a practical hardware implementation of a neural network. Initially, why implement neural networks in hardware if one could easily simulate it in software? The main reason is that the hardware implementation of artificial neural networks can take advantage of their inherent parallelism and run orders of magnitude faster than software simulations. There actually exists commercial neural network hardware like neural network processors and neurocomputers,

but they lack popularity for they are hard to find, very expensive, and very complex. From this, the main problem of this study was based; that there is no available neural network hardware that is simple yet effective in pattern recognition applications.

The main objective of this study is to design an artificial neural network controller mainly for pattern recognition applications, utilizing a field programmable gate array (FPGA) as the core component in implementing its architecture. The specific objectives of this study are to determine the accuracy and number of neurons used for the single layer perceptron neural network and the multilayer perceptron neural network; to determine if there are significant differences between the accuracies of the said neural network models; to select between the two neural network models that will be more effective to implement in hardware; to determine the accuracy of the selected neural network model in hardware; and to find out if the accuracy of the selected neural network model is consistent with the accuracy in its software implementation.

This study is significant because it simplifies the integration of neural networks to the design of systems involving pattern recognition applications. It takes advantage of the fact that neural network hardware implementations process data much faster than the neural network software implementations. The proposed artificial neural network controller is most useful to real-time pattern recognition systems such as aircraft component fault detectors, medical diagnosis systems, speech synthesizers, manufacturing process control systems, truck brake diagnosis systems, etc.

The function of the proposed artificial neural network controller shall only focus on single-digit pattern recognition. It shall have two modes of operation: training and operation mode. It will not be capable of learning while in operation mode as learning is only possible

while the neural network in training mode. Only a single neural network model with a specific learning algorithm shall be implemented in hardware. Of the numerous neural network models which are available, only two models shall be evaluated and compared: the single layer perceptron (SLP) model and the multilayer perceptron model (MLP). Only a single, specific learning algorithm shall be chosen for each of the neural network models: perceptron learning rule algorithm for SLP and backpropagation learning algorithm for MLP. The learning method that will be used is supervised learning (i.e., an input is presented to the neural network and a corresponding target response is set at the output). The number of inputs and outputs to each of the neural network models shall be restricted to 25 inputs and 25 outputs.

Methodology

The treatment of data shall consist of two major parts: one for the software simulation of the neural network models and another for the hardware implementation of the chosen neural network model. The purpose of the software simulation is to determine the performance and complexity of each model which is then used to evaluate and select which of the neural network models would be suitable for implementation in hardware. The purpose of the hardware simulation is to verify that the performance of the chosen neural network model will be consistent with the results that were gathered during its software simulation. For the software simulation, the software to be used will be MATLAB; for the hardware implementation, the National Instruments LabVIEW software and the National Instruments Digital Electronics FPGA board will be used.

The software simulation shall be the basis as to which neural network model will be implemented in hardware. The specific parameters used for performance measurement and complexity measurement are mean accuracy and number of neurons, respectively. Mean accuracy is equal to the average of individual accuracies calculated during the testing of the neural network. The number of neurons is the total number of neurons that were used in a particular neural network model. The percentage difference measures how the mean accuracies of the SLP and MLP significantly differ from each other.

The network architectures for the SLP and MLP models will be built in MATLAB using the Neural Network Toolbox. In determining the number of neurons to use for each neural network model, Equations 3.1 and 3.2 will be used:

$$\begin{aligned} \text{Number of neurons} &= \text{number of hidden layer neurons} \\ &+ \text{number of output layer neurons} \end{aligned} \quad (3.1)$$

$$\begin{aligned} \text{Number of output layer neurons} \\ &= \text{number of outputs of the neural network} \end{aligned} \quad (3.2)$$

In computing the number of hidden layer neurons (only applicable to MLP model), an incremental pruning algorithm will be used:

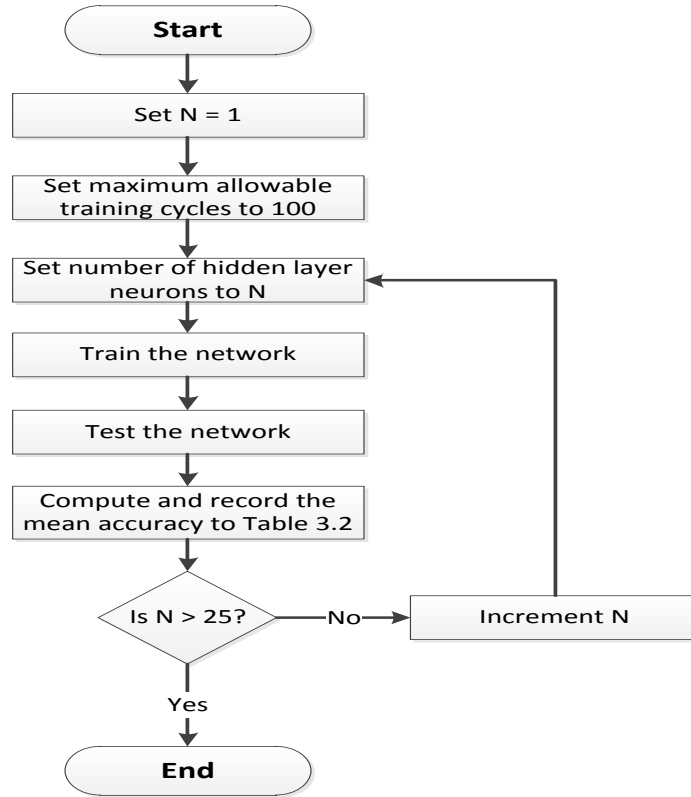


Figure 3.1. Flowchart of the incremental pruning algorithm.

The incremental pruning algorithm will be executed for 10 trials since different simulation runs on an MLP network would give different results due to its randomized initial weight and bias values. The training data patterns and test data patterns are shown in Tables 3.1 and 3.2.









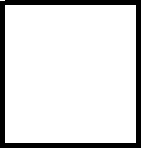


Training Patterns		
		
Digit 0	Digit 4	Digit 8
		
Digit 1	Digit 5	Digit 9
		
Digit 2	Digit 6	Null
		
Digit 3	Digit 7	

Table 3.1. Training data patterns.































Test Patterns					
					
Digit 0			Digit 5		
					
Digit 1			Digit 6		
					
Digit 2			Digit 7		
					
Digit 3			Digit 8		
					
Digit 4			Digit 9		

Table 3.2. Test data patterns.

Referring to the results gathered from the pruning algorithm, the optimal number of hidden layer neurons could now be obtained by finding the number of neurons with the highest obtained value for the average of mean accuracies in 10 trials. In case that there are two or more same values for the highest average are recorded, the one with the lesser number of neurons shall be considered. On the other hand, the highest average shall be recorded as the mean accuracy field MLP.

The constructed neural network models in MATLAB shall be trained and tested using the training and testing data patterns in Tables 3.1 and 3.2. In computing for the individual accuracies and the mean accuracy, Equations 3.3 and 3.4 shall be used respectively:

$$\text{Accuracy} = \frac{\text{number of matched squares in target and output patterns}}{\text{total number of squares in grid}} \quad (3.3)$$

$$\times 100\%$$

$$\text{Mean accuracy} = \frac{\text{summation of accuracies}}{\text{total number of accuracies}} \times 100\% \quad (3.4)$$

In computing for the percentage difference, Equation 3.5 is used:

$$\text{percentage difference} = \frac{|X_1 - X_2|}{\left(\frac{X_1 + X_2}{2}\right)} \times 100\% \quad (3.5)$$

Referring to Equation 3.5, X_1 and X_2 denote the SLP and MLP mean accuracies respectively.

For the selection of the neural network model to be implemented in hardware, the decision tree in Figure 3.2 will be followed.

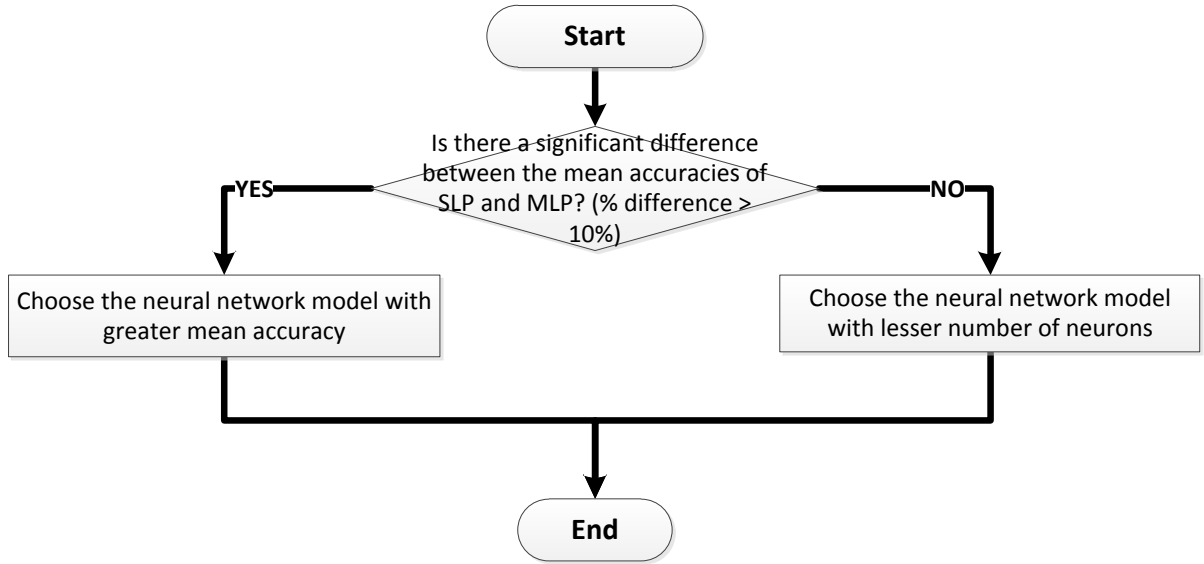


Figure 3.2. Neural network model selection decision tree.

The chosen neural network shall be designed and constructed using the software National Instruments LabVIEW. Training the neural network is also done in this software using the training data patterns in Table 3.1. The trained neural network shall be compiled and uploaded to the National Instruments Digital Electronics FPGA board.

For the hardware simulation, input patterns are presented to the chosen neural network and the resulting output patterns and accuracies are recorded using the test data patterns in Table 3.2. The percentage difference between the mean accuracies from both simulations shall be calculated using the Equation 3.5, with X_1 and X_2 equal to the mean accuracies of the software and hardware simulation, respectively. If there is no significant difference between the two simulations (percentage difference $< 10\%$), then it indicates that the results of the actual hardware simulation are consistent with the results of its software simulation.

Results and Discussion

Neural Network Model	Performance measurement	Complexity measurement
	Mean accuracy	No. of neurons
SLP	94.13 %	25
MLP	94.96 %	42
Percentage difference	0.8779 %	

Table 3.3. Performance and complexity comparison table for SLP and MLP.

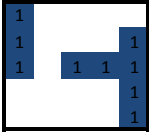
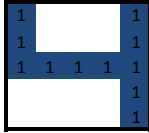
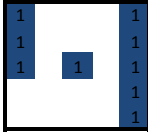
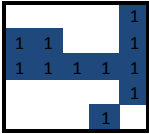
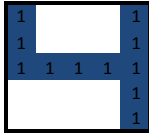
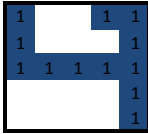
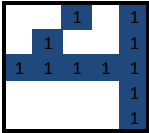
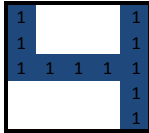
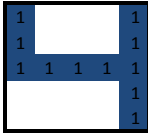
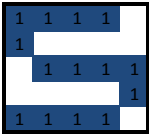
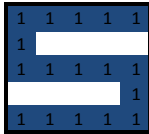
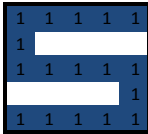
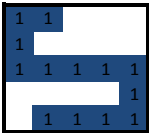
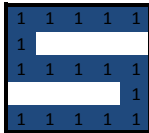
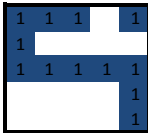
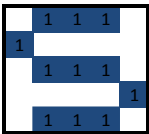
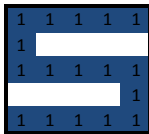
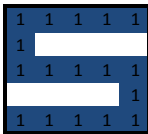
Table 3.3 shows the summary of performance and complexity measurement of the neural network models used in the software simulation. It also shows the percentage difference, which is calculated using Equation 3.5, between the computed mean accuracy of both models. The mean accuracy of SLP was derived from the SLP accuracy measurement for software simulation shown in Table 3.5, while the mean accuracy of MLP was the higher average obtained in the incremental pruning algorithm. Table 3.4 shows the result of the incremental pruning algorithm done in MLP for software simulation. The highest obtained value for the average mean of accuracies in 10 trials is 94.96 %. From this result, the most favorable number of hidden layer neurons in MLP is 17. Table 3.6 shows the MLP accuracy measurement for software simulation.

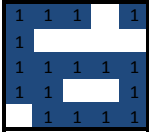
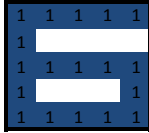
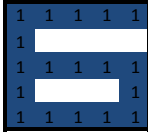
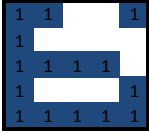
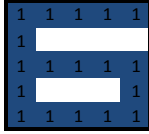
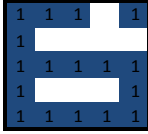
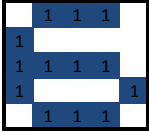
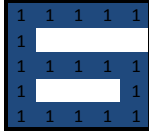
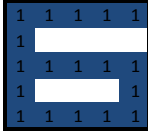
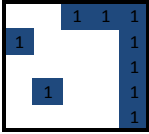
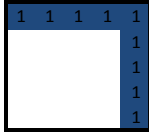
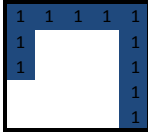
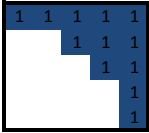
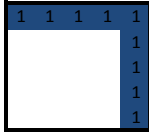
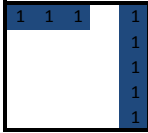
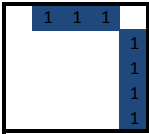
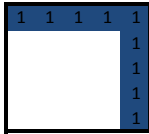
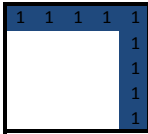
Number of Neurons	Mean Accuracies										
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Average
1	79.60%	81.87%	86.13%	80.53%	85.20%	86.93%	85.87%	85.47%	85.60%	85.60%	84.28%
2	87.33%	83.73%	88.27%	89.20%	89.47%	85.73%	84.13%	88.40%	83.87%	86.93%	86.71%
3	90.53%	90.00%	92.80%	85.47%	88.80%	89.87%	88.67%	87.60%	91.87%	89.20%	89.48%
4	91.07%	91.20%	91.47%	87.60%	94.27%	89.20%	89.47%	93.07%	91.60%	89.73%	90.87%
5	90.80%	91.33%	88.27%	92.67%	92.00%	89.73%	84.80%	92.00%	93.60%	95.33%	91.05%
6	93.73%	92.27%	92.00%	93.20%	91.07%	92.00%	87.87%	91.47%	90.40%	92.67%	91.67%
7	90.93%	92.53%	90.53%	93.20%	89.20%	90.93%	93.73%	93.60%	93.60%	94.93%	92.32%
8	93.87%	95.73%	94.80%	93.33%	92.27%	92.67%	91.20%	91.20%	90.53%	94.53%	93.01%
9	94.13%	93.60%	90.67%	92.53%	93.07%	91.47%	93.20%	94.00%	94.27%	90.13%	92.71%
10	92.67%	93.60%	94.67%	94.93%	91.73%	92.40%	92.53%	95.07%	92.27%	95.87%	93.57%
11	91.07%	92.80%	92.27%	90.53%	92.67%	95.07%	93.20%	94.93%	90.40%	95.07%	92.80%
12	94.13%	93.33%	91.47%	92.53%	93.73%	93.60%	90.93%	93.20%	92.13%	94.40%	92.95%
13	91.73%	96.13%	92.80%	93.60%	92.80%	94.13%	94.40%	94.40%	93.60%	94.00%	93.76%
14	92.93%	93.87%	90.00%	93.47%	93.80%	93.47%	94.67%	93.87%	94.53%	94.00%	93.46%
15	94.27%	88.80%	95.20%	95.33%	93.73%	95.20%	95.47%	93.20%	93.33%	93.87%	93.84%
16	91.47%	92.80%	90.67%	92.40%	94.00%	92.93%	94.53%	92.80%	94.40%	94.93%	93.09%
17	95.47%	93.33%	96.00%	96.00%	94.13%	94.27%	95.07%	95.73%	94.00%	95.60%	94.96%
18	89.07%	94.40%	94.40%	92.67%	94.67%	94.67%	93.87%	94.40%	91.60%	96.80%	93.66%
19	95.73%	96.13%	91.33%	94.53%	92.80%	93.20%	94.40%	94.13%	94.13%	92.80%	93.92%
20	95.07%	95.47%	94.27%	93.87%	91.87%	90.53%	94.93%	95.73%	95.07%	93.60%	94.04%
21	93.20%	95.20%	91.60%	95.60%	94.40%	95.47%	94.00%	95.20%	95.33%	90.53%	94.05%
22	94.27%	94.53%	96.93%	92.67%	94.93%	94.27%	90.27%	94.80%	93.20%	94.40%	94.03%
23	95.20%	92.93%	91.87%	94.13%	95.20%	95.73%	95.07%	91.73%	94.93%	94.67%	94.15%
24	94.13%	92.80%	91.20%	93.20%	94.13%	94.40%	92.67%	95.20%	94.40%	93.60%	93.57%
25	94.13%	95.33%	91.73%	92.53%	93.33%	94.53%	93.73%	94.80%	94.93%	94.80%	93.98%
Maximum Average											94.96%

Table 3.4. MLP Pruning Table.

Neural Network Model	Test Pattern	Target Pattern	Output Pattern	Accuracy
Single Layer Perceptron with Perceptron Learning Rule				96.00%
				92.00%
				100.00%
				100.00%
				92.00%
				100.00%

			92.00%
			92.00%
			96.00%
			92.00%
			96.00%
			96.00%

			92.00%
			96.00%
			100.00%
			100.00%
			80.00%
			100.00%

			100.00%
			96.00%
			100.00%
			92.00%
			96.00%
			100.00%

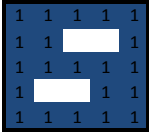
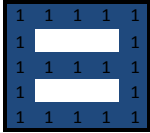
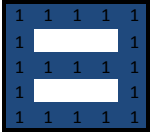
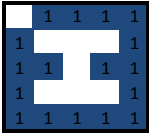
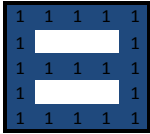
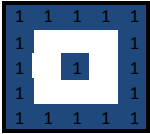
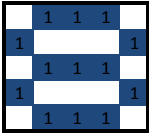
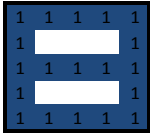
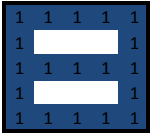
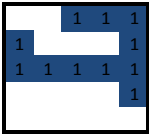
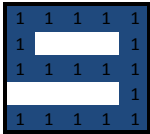
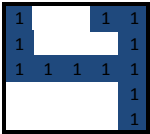
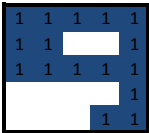
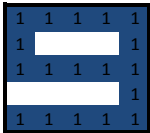
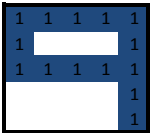
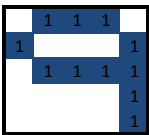
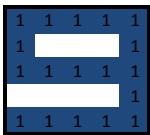
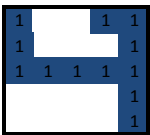
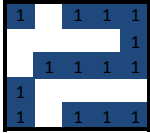
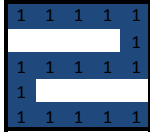
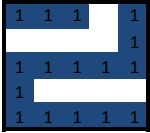
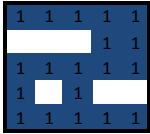
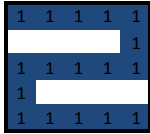
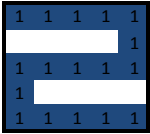
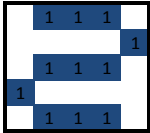
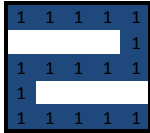
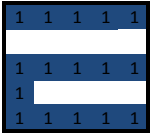
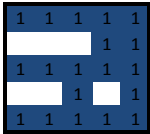
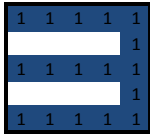
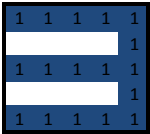
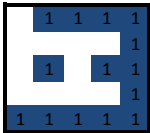
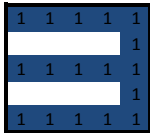
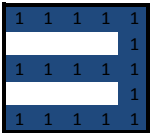
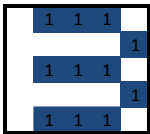
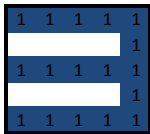
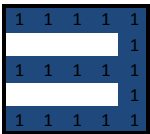
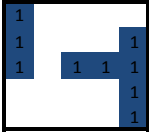
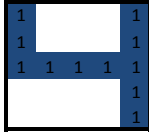
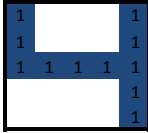
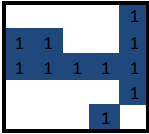
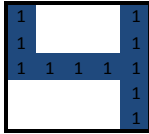
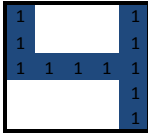
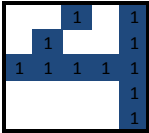
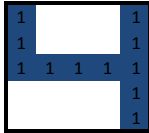
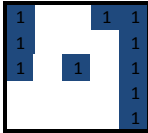
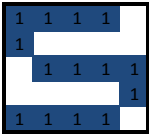
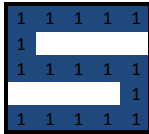
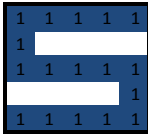
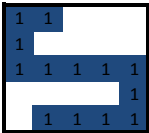
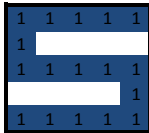
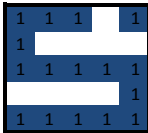
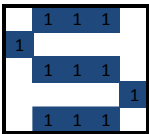
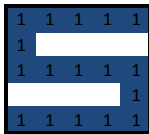
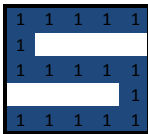
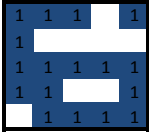
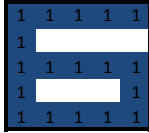
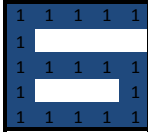
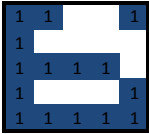
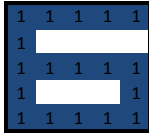
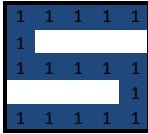
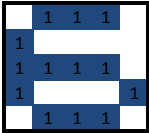
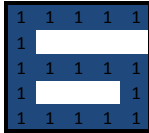
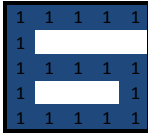
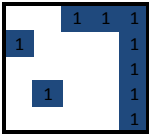
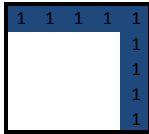
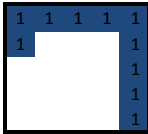
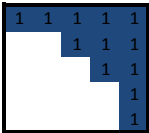
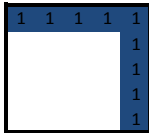
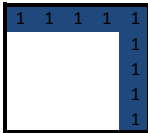
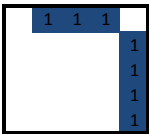
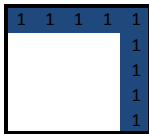
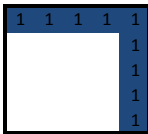
			100.00%
			92.00%
			100.00%
			76.00%
			84.00%
			76.00%
Mean Accuracy			94.13%

Table 3.5. SLP accuracy measurement table for software simulation.

Neural Network Model	Test Pattern	Target Pattern	Output Pattern	Accuracy
Multilayer Perceptron with Backpropagation Learning				88.00%
				88.00%
				100.00%
				100.00%
				84.00%
				100.00%

			96.00%
			100.00%
			96.00%
			100.00%
			100.00%
			100.00%

			100.00%
			100.00%
			88.00%
			100.00%
			96.00%
			100.00%

			100.00%
			96.00%
			100.00%
			96.00%
			100.00%
			100.00%

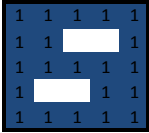
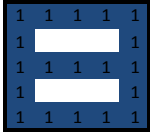
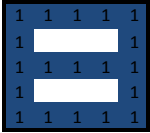
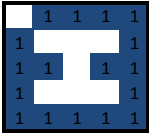
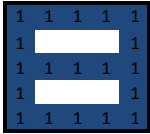
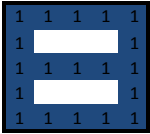
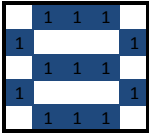
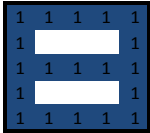
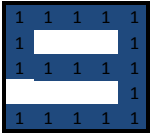
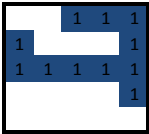
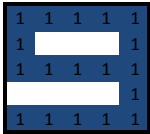
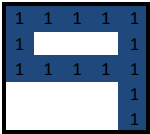
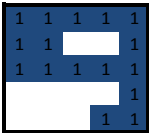
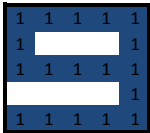
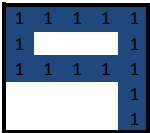
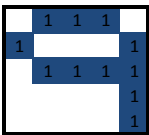
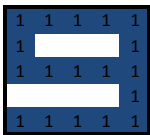
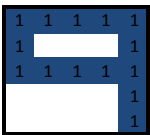
			100.00%
			100.00%
			96.00%
			84.00%
			84.00%
			84.00%
Mean Accuracy			95.87%

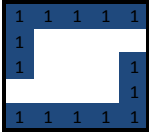
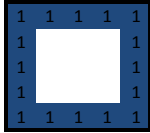
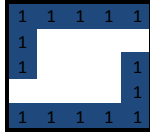
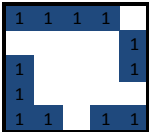
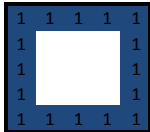
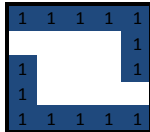
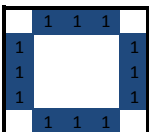
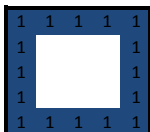
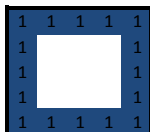
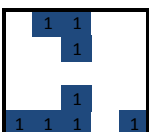
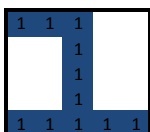
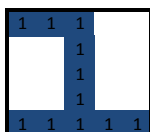
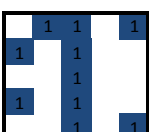
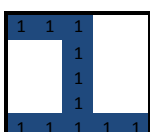
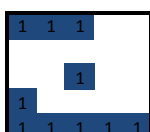
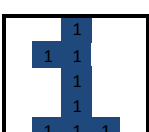

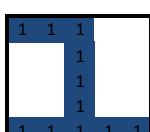
Table 3.6. Sample MLP accuracy measurement table for software simulation.

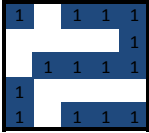
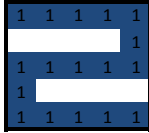
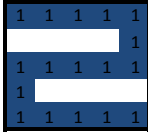
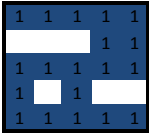
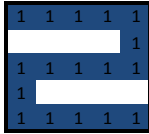
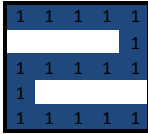
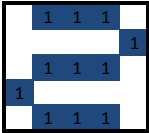
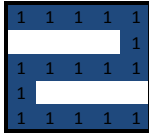
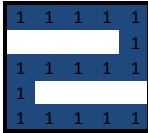
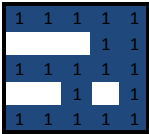
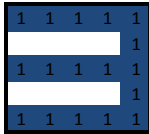

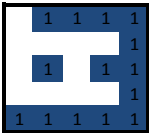
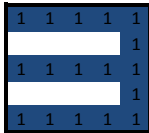
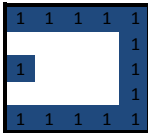
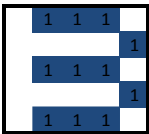


During the software simulation, three test patterns were presented for each target pattern. Accuracy for every test pattern is computed using Equation 3.3. The mean accuracy is then calculated using equation 3.4 in both neural network models.

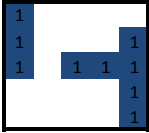
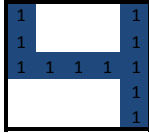
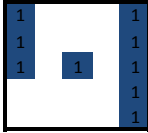
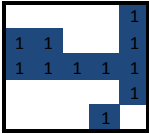
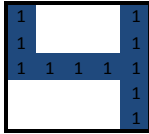
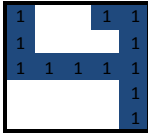
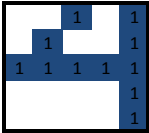
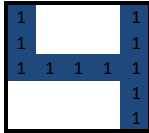
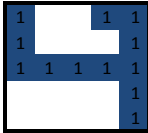
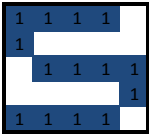
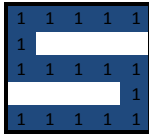
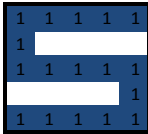
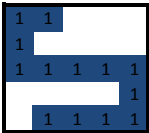
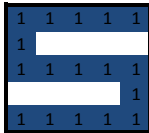
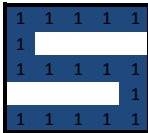
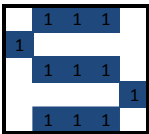
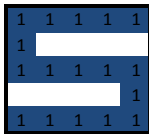
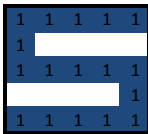
Table 3.3 shows both neural network models resulted to very high mean accuracies. And since the percentage difference between the mean accuracies of both neural network models is insignificant, the total number of neurons used per neural network model was

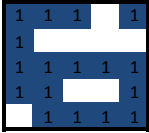
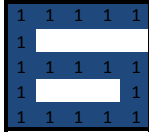
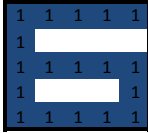
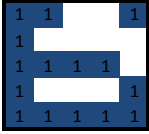
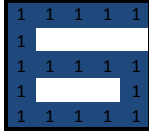
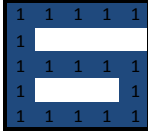
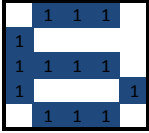
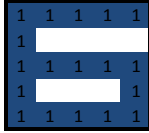
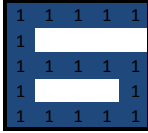
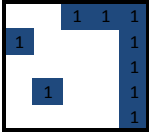
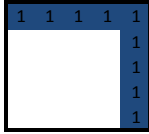
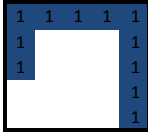
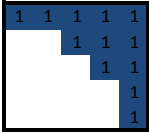
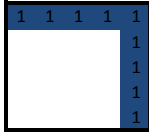
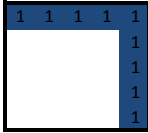
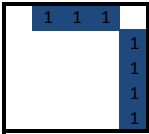
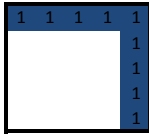
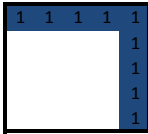
chosen as the criterion in choosing the model that will be used in hardware implementation.

With the smaller number of neurons, the SLP was the model chosen for hardware implementation. Table 3.7 shows the SLP accuracy measurement for hardware simulation.

Neural Network Model	Test Pattern	Target Pattern	Output Pattern	Accuracy
Single Layer Perceptron with Perceptron Learning Rule				92.00%
				92.00%
				100.00%
				100.00%
				88.00%
				100.00%

			100.00%
			100.00%
			100.00%
			96.00%
			88.00%
			100.00%

			92.00%
			96.00%
			96.00%
			100.00%
			100.00%
			100.00%

			100.00%
			100.00%
			100.00%
			92.00%
			100.00%
			100.00%

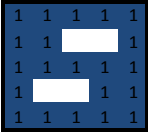
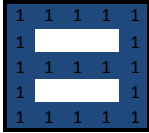
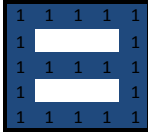
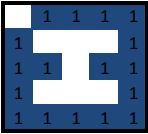
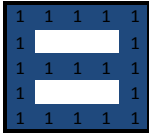
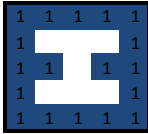
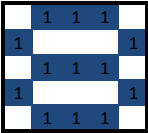
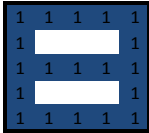
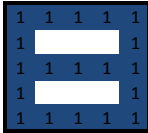
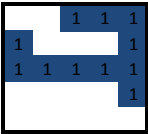
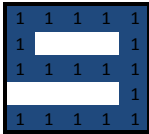
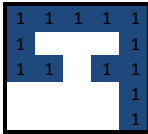
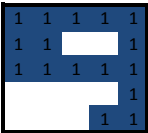
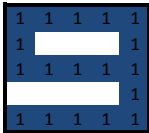
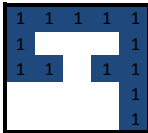
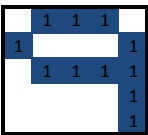
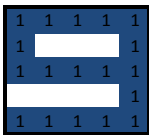
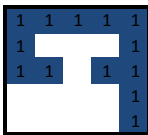
			100.00%
			96.00%
			100.00%
			80.00%
			80.00%
			80.00%
Mean Accuracy			95.60%

Table 3.7. SLP accuracy measurement table for hardware simulation.

Simulation Type	Performance measurement
	Mean accuracy
Software simulation	94.13 %
Hardware simulation	95.60 %
Percentage difference	1.5496 %

Table 3.8. Performace comparison table for the hardware and software simulation.

The hardware simulation resulted to 95.60% mean accuracy. Similar with the results gathered in software simulation, the outputs generated by the implemented hardware were almost similar to the respective target patterns. Table 3.8 shows the percentage difference between the two simulations done. From the results gathered, it can be said that the outcome of the hardware implementation is consistent with the outcome of the software implementation.

Conclusion

From the results gathered during the software simulation, there is no significant difference between the mean accuracies of both SLP and MLP in software simulation. The SLP resulted to 94.13% mean accuracy, while the MLP resulted to 94.96% mean accuracy. In both neural network models, the generated outputs, most of the time, matched their respective target patterns. Since the percentage difference between the mean accuracies of these two neural network models did not exceed 10%, the complexity of the model was chosen as the criterion for choosing the model that will be used in hardware implementation. In evaluating the complexity of the model, the total number of neurons per model is considered. The SLP is composed of 25 output layer neurons, while the MLP is composed of 17 hidden layer neurons and 25 output layer neurons. With the smaller total number of neurons in its structure, the SLP was chosen as the model that will be applied in the hardware.

The hardware simulation resulted to 95.60% mean accuracy. The percentage difference between the mean accuracy gathered in both software and hardware simulation is

1.5496%. With this outcome, the researchers have concluded that the result of the hardware implementation is consistent with the result of the software implementation.

From the results reported, the researchers conclude that an artificial neural network controller for single digit pattern recognition using FPGA was successfully implemented. The neural network model implemented in the hardware is simple enough to fit in the FPGA yet effective in recognizing input patterns presented to it. The whole system is able to learn single digit patterns during training mode and to recognize perfect and corrupted single digit patterns accurately during operation mode.

References

- Ahadi, S.M., A. Pourmohammad (2009). Using Single-Layer neural network for recognition of isolated handwritten Persian digits. *In Proceedings International Conference on Communications and Signal Processing on Information ICICS*, Volume 7, 1-4.
- Bhattacharyya, S.K., N.N. Biswas (1991). Tuning capability of binary perceptron. *Electronics Letters*, Volume 27 (13), 1206-1207.
- Badgero, M.L. (1994). Digitizing artificial neural networks. Neural Networks, 1994. *In Proceedings IEEE International Conference on IEEE World Congress on Computational Intelligence*, Volume 6, 3986-3989.
- Barton, J.B., D.R. Collins, M.T. Gately, P.A. Penz (1991). Neural network hardware: what does the future hold?. *In Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Volume 1, 455-467.
- Beale, M.H., H.B. Demuth, M.T. Hagan (1996). Neural Network Design. Boston: PWS, Print.
- Besbes, K., H. Faiedh, Z. Gafsi, K. Torki (2004). Digital hardware implementation of a neural network used for classification. *In Proceedings The 16th International Conference on Microelectronics*, 551- 554.
- Buyukaksoy, G., H.O. Gulcur (1992). Identification of different types of leucocytes in dried blood smears using neural networks. *In Proceedings of the International Biomedical Engineering Days*, 203-206.

- DasGupta, S., Z. Delalic , F. Kampf, P. Koch, K. Roy, M. Sullivan (1989). Digital implementation of a neural network. *In Proceedings International Joint Conference on Neural Networks IJCNN*, Volume 2, 581.
- Fuller, R. (2010). The Perception Learning Rule – Tutorial. Institute for Advanced Management Systems Research Department of Information Technologies.
- Graupe D. (2007). Principles of Artificial Neural Networks, 2nd edition, World Scientific Publishing Co. Pte. Ltd., Singapore.
- Haykin, S. (1999). Neural Networks - A Comprehensive Foundation, 2nd edition, Pearson Education Pte. Ltd, India.
- Hobson, R.S., H. Raju, P.A. Wetzel (2001). Neural network for visual search classification. *In Proceedings of the 23rd Annual International Conference of the IEEE on Engineering in Medicine and Biology Society*, Volume 3, 2737- 2739.
- Introduction to artificial neural networks. *In Proceedings Electronic Technology Directions to the Year 2000*, May 1995, 36-62.
- Kochan, V., T. Laopoulos, A. Sachenko, V. Turchenko (2001). The new method of historical sensor data integration using neural networks. *In Proceedings International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, 21-24.
- National Instruments Inc. (2005). NI LabView FPGA User's Guide.
- Tarassenko, L. (1998). A Guide to Neural Computing Applications. London: Arnold, Print.
- The MathWorks, Inc. (2010). Neural Network Toolbox™ 7 User's Guide.
- U. Meyer-Baese (2007). Digital Signal Processing with Field Programmable Gate Arrays Third Edition, Springer Berlin Heidelberg New York.
- Veelenturf L.P.J. (1995). Analysis and Applications of Artificial Neural Networks, Prentice Hall International (UK) Limited, Hertfordshire.

Chapter 4

CONCLUSION

From the results gathered during the software simulation, there is no significant difference between the mean accuracies of both SLP and MLP in software simulation. The SLP resulted to 94.13% mean accuracy, while the MLP resulted to 94.96% mean accuracy. In both neural network models, the generated outputs, most of the time, matched their respective target patterns. Since the percentage difference between the mean accuracies of these two neural network models did not exceed 10%, the complexity of the model was chosen as the criterion for choosing the model that will be used in hardware implementation. In evaluating the complexity of the model, the total number of neurons per model is considered. The SLP is composed of 25 output layer neurons, while the MLP is composed of 17 hidden layer neurons and 25 output layer neurons. With the smaller total number of neurons in its structure, the SLP was chosen as the model that will be applied in the hardware.

The hardware simulation resulted to 95.60% mean accuracy. The percentage difference between the mean accuracy gathered in both software and hardware simulation is 1.5496%. With this outcome, the researchers have concluded that the result of the hardware implementation is consistent with the result of the software implementation.

From the results reported, the researchers conclude that an artificial neural network controller for single digit pattern recognition using FPGA was successfully implemented. The neural network model implemented in the hardware is simple enough to fit in the FPGA yet effective in recognizing input patterns presented to it. The whole system is able to learn

single digit patterns during training mode and to recognize perfect and corrupted single digit patterns accurately during operation mode.

Chapter 5

RECOMMENDATION

Low-price FPGA development boards can be used other than the National Instruments Digital Electronics FPGA to minimize the cost of the whole system. Trenz Electronic, Simple Solutions, Digilent, and Atmark Techno are some of the companies that manufacture low cost FPGA development boards.

More complicated patterns can be taught and trained to the neural network. This involves redesigning the architecture of the whole neural network to accommodate several complex patterns.

Lastly, several applications of neural networks other than pattern recognition could be employed in hardware to suit different needs in many fields of science and engineering. Some useful applications include function approximation, data segmentation, data compression, data mining, optimization, complex mapping, signal filtering, and many more.

REFERENCES

- Ahadi, S.M., A. Pourmohammad (2009). Using Single-Layer neural network for recognition of isolated handwritten Persian digits. *In Proceedings International Conference on Communications and Signal Processing on Information ICICS*, Volume 7, 1-4.
- Bhattacharyya, S.K., N.N. Biswas (1991). Tuning capability of binary perceptron. *Electronics Letters*, Volume 27 (13), 1206-1207.
- Badgero, M.L. (1994). Digitizing artificial neural networks. Neural Networks, 1994. *In Proceedings IEEE International Conference on IEEE World Congress on Computational Intelligence*, Volume 6, 3986-3989.
- Barton, J.B., D.R. Collins, M.T. Gately, P.A. Penz (1991). Neural network hardware: what does the future hold?. *In Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Volume 1, 455-467.
- Beale, M.H., H.B. Demuth, M.T. Hagan (1996). Neural Network Design. Boston: PWS, Print.
- Besbes, K., H. Faiedh, Z. Gafsi, K. Torki (2004). Digital hardware implementation of a neural network used for classification. *In Proceedings The 16th International Conference on Microelectronics*, 551- 554.
- Buyukaksoy, G., H.O. Gulcur (1992). Identification of different types of leucocytes in dried blood smears using neural networks. *In Proceedings of the International Biomedical Engineering Days*, 203-206.
- DasGupta, S., Z. Delalic , F. Kampf, P. Koch, K. Roy, M. Sullivan (1989). Digital implementation of a neural network. *In Proceedings International Joint Conference on Neural Networks IJCNN*, Volume 2, 581.
- Fuller, R. (2010). The Perception Learning Rule – Tutorial. Institute for Advanced Management Systems Research Department of Information Technologies.
- Graupe D. (2007). Principles of Artificial Neural Networks, 2nd edition, World Scientific Publishing Co. Pte. Ltd., Singapore.
- Haykin, S. (1999). Neural Networks - A Comprehensive Foundation, 2nd edition, Pearson Education Pte. Ltd, India.
- Hobson, R.S., H. Raju, P.A. Wetzel (2001). Neural network for visual search classification. *In Proceedings of the 23rd Annual International Conference of the IEEE on Engineering in Medicine and Biology Society*, Volume 3, 2737- 2739.

Introduction to artificial neural networks. *In Proceedings Electronic Technology Directions to the Year 2000*, May 1995, 36-62.

Kochan, V., T. Laopoulos, A. Sachenko, V. Turchenko (2001). The new method of historical sensor data integration using neural networks. *In Proceedings International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, 21-24.

National Instruments Inc. (2005). NI LabView FPGA User's Guide.

Tarassenko, L. (1998). A Guide to Neural Computing Applications. London: Arnold, Print.

The MathWorks, Inc. (2010). Neural Network Toolbox™ 7 User's Guide.

U. Meyer-Baese (2007). Digital Signal Processing with Field Programmable Gate Arrays Third Edition, Springer Berlin Heidelberg New York.

Veelenturf L.P.J. (1995). Analysis and Applications of Artificial Neural Networks, Prentice Hall International (UK) Limited, Hertfordshire.

APPENDICES

APPENDIX A

Neural Network Software Simulation Program Source Code

```
function varargout = NNSoftwareSim(varargin)
% NNSOFTWARESIM M-file for NNSoftwareSim.fig
%     NNSOFTWARESIM, by itself, creates a new NNSOFTWARESIM or raises the
existing
%     singleton*.
%
%     H = NNSOFTWARESIM returns the handle to a new NNSOFTWARESIM or the
handle to
%     the existing singleton*.
%
%     NNSOFTWARESIM('CALLBACK',hObject,eventData,handles,...) calls the
local
%     function named CALLBACK in NNSOFTWARESIM.M with the given input
arguments.
%
%     NNSOFTWARESIM('Property','Value',...) creates a new NNSOFTWARESIM
or raises the
%     existing singleton*. Starting from the left, property value pairs
are
%     applied to the GUI before NNSoftwareSim_OpeningFcn gets called. An
%     unrecognized property name or invalid value makes property
application
%     stop. All inputs are passed to NNSoftwareSim_OpeningFcn via
varargin.
%
%     *See GUI Options on GUIDE's Tools menu. Choose "GUI allows only
one
%     instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help NNSoftwareSim

% Last Modified by GUIDE v2.5 15-Aug-2011 11:11:31

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                  'gui_Singleton',   gui_Singleton, ...
                  'gui_OpeningFcn', @NNSoftwareSim_OpeningFcn, ...
                  'gui_OutputFcn',  @NNSoftwareSim_OutputFcn, ...
                  'gui_LayoutFcn',   [] , ...
                  'gui_Callback',    []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
```

```

end
% End initialization code - DO NOT EDIT

% --- Executes just before NNSoftwareSim is made visible.
function NNSoftwareSim_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles     structure with handles and user data (see GUIDATA)
% varargin    command line arguments to NNSoftwareSim (see VARARGIN)

% Choose default command line output for NNSoftwareSim
handles.output = hObject;

% Update handles structure
set(handles.mirrorCheckbox, 'Value', 1);
handles.inputMatrix = zeros(25,1);
handles.targetMatrix = zeros(25,1);
handles.inputButtonHandles = [handles.IA1; handles.IA2; handles.IA3;
handles.IA4; handles.IA5;
handles.IB1; handles.IB2; handles.IB3;
handles.IB4; handles.IB5;
handles.IC1; handles.IC2; handles.IC3;
handles.IC4; handles.IC5;
handles.ID1; handles.ID2; handles.ID3;
handles.ID4; handles.ID5;
handles.IE1; handles.IE2; handles.IE3;
handles.IE4; handles.IE5];

handles.targetButtonHandles = [handles.TA1; handles.TA2; handles.TA3;
handles.TA4; handles.TA5;
handles.TB1; handles.TB2; handles.TB3;
handles.TB4; handles.TB5;
handles.TC1; handles.TC2; handles.TC3;
handles.TC4; handles.TC5;
handles.TD1; handles.TD2; handles.TD3;
handles.TD4; handles.TD5;
handles.TE1; handles.TE2; handles.TE3;
handles.TE4; handles.TE5];

handles.outputButtonHandles = [handles.OA1; handles.OA2; handles.OA3;
handles.OA4; handles.OA5;
handles.OB1; handles.OB2; handles.OB3;
handles.OB4; handles.OB5;
handles.OC1; handles.OC2; handles.OC3;
handles.OC4; handles.OC5;
handles.OD1; handles.OD2; handles.OD3;
handles.OD4; handles.OD5;
handles.OE1; handles.OE2; handles.OE3;
handles.OE4; handles.OE5];
handles.column = 1;
handles.columnSelector = 1;
handles.k = 0;
handles.SLP = network;
handles.MLP = network;

```

```

set(handles.leftPB, 'Enable', 'off');
set(handles.rightPB, 'Enable', 'off');
set(handles.deletePB, 'Enable', 'off');
set(handles.trainPB, 'Enable', 'off');
set(handles.outputPB, 'Enable', 'off');
set(handles.accuracyText, 'Enable', 'off');
set(handles.slpRB, 'Value', 1);
set(handles.mlpRB, 'Value', 0);
set(handles.numPatternsText, 'String', '0');
set(handles.accuracyText, 'String', '100%');
guidata(hObject, handles);

% UIWAIT makes NNSoftwareSim wait for user response (see UIRESUME)
% uiwait(handles.mainFig);

% --- Outputs from this function are returned to the command line.
function varargout = NNSoftwareSim_OutputFcn(hObject, eventdata, handles)
% varargout cell array for returning output args (see VARARGOUT);
% hObject handle to figure
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

function reflectToInput(handles)
for R = 1:25
    val1 = handles.inputMatrix(R, handles.columnSelector);
    set(handles.inputButtonHandles(R,1), 'Value', val1);
    if (val1 == 1)
        set(handles.inputButtonHandles(R,1), 'BackgroundColor',
'White');
    else
        set(handles.inputButtonHandles(R,1), 'BackgroundColor',
'Black');
    end
    val2 = handles.targetMatrix(R, handles.columnSelector);
    set(handles.targetButtonHandles(R,1), 'Value', val2);
    if (val2 == 1)
        set(handles.targetButtonHandles(R,1), 'BackgroundColor',
'White');
    else
        set(handles.targetButtonHandles(R,1), 'BackgroundColor',
'Black');
    end
end

% ---
function populateInputMatrix(hSrc, hDest, row, handles)

```

```

h = get(hSrc, 'Value');
if (h == 1)
    handles.inputMatrix(row, handles.column) = 1;
    set(hSrc, 'BackgroundColor', 'White');
else
    handles.inputMatrix(row, handles.column) = 0;
    set(hSrc, 'BackgroundColor', 'Black');
end

if (get(handles.mirrorCheckbox, 'Value') == 1)
    val = get(hSrc, 'Value');
    set(hDest, 'Value', val);
    col = get(hSrc, 'BackgroundColor');
    set(hDest, 'BackgroundColor', col);
    handles.targetMatrix(row, handles.column) = handles.inputMatrix(row,
handles.column);
end

handles.columnSelector = handles.column;
handles.k = 0;
set(handles.rightPB, 'Enable', 'off');
set(handles.deletePB, 'Enable', 'off');
set(handles.accuracyText, 'Enable', 'off');
if (handles.column > 2)
    set(handles.leftPB, 'Enable', 'on');
end
guidata(hSrc, handles);

function populateTargetMatrix(hSrc, hDest, row, handles)

h = get(hSrc, 'Value');
if (h == 1)
    handles.targetMatrix(row, handles.column) = 1;
    set(hSrc, 'BackgroundColor', 'White');
else
    handles.targetMatrix(row, handles.column) = 0;
    set(hSrc, 'BackgroundColor', 'Black');
end

if (get(handles.mirrorCheckbox, 'Value') == 1)
    val = get(hSrc, 'Value');
    set(hDest, 'Value', val);
    col = get(hSrc, 'BackgroundColor');
    set(hDest, 'BackgroundColor', col);
    handles.inputMatrix(row, handles.column) = handles.targetMatrix(row,
handles.column);
end
handles.columnSelector = handles.column;
handles.k = 0;
set(handles.rightPB, 'Enable', 'off');
set(handles.deletePB, 'Enable', 'off');
set(handles.accuracyText, 'Enable', 'off');
if (handles.column > 2)
    set(handles.leftPB, 'Enable', 'on');
end
guidata(hSrc, handles);

```

```

% --- Executes on button press in IA1.
function IA1_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IA1, handles.TA1, 1, handles);

% --- Executes on button press in IA2.
function IA2_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IA2, handles.TA2, 2, handles);

% --- Executes on button press in IA3.
function IA3_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IA3, handles.TA3, 3, handles);

% --- Executes on button press in IA4.
function IA4_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IA4, handles.TA4, 4, handles);

% --- Executes on button press in IA5.
function IA5_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IA5, handles.TA5, 5, handles);

% --- Executes on button press in IB1.
function IB1_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IB1, handles.TB1, 6, handles);

% --- Executes on button press in IB2.
function IB2_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IB2, handles.TB2, 7, handles);

% --- Executes on button press in IB3.
function IB3_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IB3, handles.TB3, 8, handles);

% --- Executes on button press in IB4.
function IB4_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IB4, handles.TB4, 9, handles);

% --- Executes on button press in IB5.
function IB5_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IB5, handles.TB5, 10, handles);

% --- Executes on button press in IC1.
function IC1_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IC1, handles.TC1, 11, handles);

% --- Executes on button press in IC2.
function IC2_Callback(hObject, eventdata, handles)

```

```

populateInputMatrix(handles.IC2, handles.TC2, 12, handles);

% --- Executes on button press in IC3.
function IC3_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IC3, handles.TC3, 13, handles);

% --- Executes on button press in IC4.
function IC4_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IC4, handles.TC4, 14, handles);

% --- Executes on button press in IC5.
function IC5_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IC5, handles.TC5, 15, handles);

% --- Executes on button press in ID1.
function ID1_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.ID1, handles.TD1, 16, handles);

% --- Executes on button press in ID2.
function ID2_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.ID2, handles.TD2, 17, handles);

% --- Executes on button press in ID3.
function ID3_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.ID3, handles.TD3, 18, handles);

% --- Executes on button press in ID4.
function ID4_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.ID4, handles.TD4, 19, handles);

% --- Executes on button press in ID5.
function ID5_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.ID5, handles.TD5, 20, handles);

% --- Executes on button press in IE1.
function IE1_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IE1, handles.TE1, 21, handles);

% --- Executes on button press in IE2.
function IE2_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IE2, handles.TE2, 22, handles);

% --- Executes on button press in IE3.
function IE3_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IE3, handles.TE3, 23, handles);

% --- Executes on button press in IE4.
function IE4_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IE4, handles.TE4, 24, handles);
% --- Executes on button press in IE5.
function IE5_Callback(hObject, eventdata, handles)
populateInputMatrix(handles.IE5, handles.TE5, 25, handles);
% --- Executes on button press in TA1.
function TA1_Callback(hObject, eventdata, handles)

```

```

populateTargetMatrix(handles.TA1, handles.IA1, 1, handles);
% --- Executes on button press in TA2.
function TA2_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TA2, handles.IA2, 2, handles);
% --- Executes on button press in TA3.
function TA3_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TA3, handles.IA3, 3, handles);
% --- Executes on button press in TA4.
function TA4_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TA4, handles.IA4, 4, handles);
% --- Executes on button press in TA5.
function TA5_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TA5, handles.IA5, 5, handles);
% --- Executes on button press in TB1.
function TB1_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TB1, handles.IB1, 6, handles);
% --- Executes on button press in TB2.
function TB2_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TB2, handles.IB2, 7, handles);
% --- Executes on button press in TB3.
function TB3_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TB3, handles.IB3, 8, handles);
% --- Executes on button press in TB4.
function TB4_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TB4, handles.IB4, 9, handles);
% --- Executes on button press in TB5.
function TB5_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TB5, handles.IB5, 10, handles);
% --- Executes on button press in TC1.
function TC1_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TC1, handles.IC1, 11, handles);
% --- Executes on button press in TC2.
function TC2_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TC2, handles.IC2, 12, handles);
% --- Executes on button press in TC3.
function TC3_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TC3, handles.IC3, 13, handles);
% --- Executes on button press in TC4.
function TC4_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TC4, handles.IC4, 14, handles);
% --- Executes on button press in TC5.
function TC5_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TC5, handles.IC5, 15, handles);
% --- Executes on button press in TD1.
function TD1_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TD1, handles.ID1, 16, handles);
% --- Executes on button press in TD2.
function TD2_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TD2, handles.ID2, 17, handles);
% --- Executes on button press in TD3.
function TD3_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TD3, handles.ID3, 18, handles);
% --- Executes on button press in TD4.
function TD4_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TD4, handles.ID4, 19, handles);
% --- Executes on button press in TD5.
function TD5_Callback(hObject, eventdata, handles)

```



```

populateTargetMatrix(handles.TD5, handles.ID5, 20, handles);
% --- Executes on button press in TE1.
function TE1_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TE1, handles.IE1, 21, handles);
% --- Executes on button press in TE2.
function TE2_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TE2, handles.IE2, 22, handles);
% --- Executes on button press in TE3.
function TE3_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TE3, handles.IE3, 23, handles);
% --- Executes on button press in TE4.
function TE4_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TE4, handles.IE4, 24, handles);
% --- Executes on button press in TE5.
function TE5_Callback(hObject, eventdata, handles)
populateTargetMatrix(handles.TE5, handles.IE5, 25, handles);
% --- Executes on button press in mirrorCheckbox.
function mirrorCheckbox_Callback(hObject, eventdata, handles)
if (get(handles.mirrorCheckbox, 'Value') == 1)
    for R = 1:25
        set(handles.targetButtonHandles(R,1), 'Value',
get(handles.inputButtonHandles(R,1), 'Value'));
        set(handles.targetButtonHandles(R,1), 'BackgroundColor',
get(handles.inputButtonHandles(R,1), 'BackgroundColor'));
        handles.targetMatrix(R, handles.column) = handles.inputMatrix(R,
handles.column);
    end
    guidata(hObject, handles);
end

% --- Executes on button press in storePB.
function storePB_Callback(hObject, eventdata, handles)
for R = 1:25
    val1 = get(handles.inputButtonHandles(R, 1), 'Value');
    handles.inputMatrix(R, handles.column) = val1;
    val2 = get(handles.targetButtonHandles(R, 1), 'Value');
    handles.targetMatrix(R, handles.column) = val2;
end
handles.inputMatrix = [handles.inputMatrix,
handles.inputMatrix(1:25,handles.column)];
handles.targetMatrix = [handles.targetMatrix,
handles.targetMatrix(1:25,handles.column)];
handles.column = handles.column + 1;
handles.columnSelector = handles.column;
handles.k = 1;
set(handles.numPatternsText, 'String', num2str(handles.column-1));
set(handles.leftPB, 'Enable', 'on');
set(handles.rightPB, 'Enable', 'off');
set(handles.deletePB, 'Enable', 'on');
set(handles.trainPB, 'Enable', 'on');
set(handles.accuracyText, 'Enable', 'off');
guidata(hObject, handles);

% --- Executes on button press in deletePB.
function deletePB_Callback(hObject, eventdata, handles)

```

```

if (handles.columnSelector == handles.column)
    handles.columnSelector = handles.column - 1;
end
handles.inputMatrix(:, handles.columnSelector) = [];
handles.targetMatrix(:, handles.columnSelector) = [];

if (handles.column == 2)
    for R = 1:25
        handles.inputMatrix(R, handles.column - 1) = 0;
        handles.targetMatrix(R, handles.column - 1) = 0;
    end
elseif (handles.columnSelector == handles.column - 1)
    handles.columnSelector = handles.columnSelector - 1;
    if (handles.columnSelector == 1)
        set(handles.leftPB, 'Enable', 'off');
    end
end

reflectToInput(handles);
handles.column = handles.column - 1;
if (handles.column == 1)
    set(handles.deletePB, 'Enable', 'off');
    set(handles.leftPB, 'Enable', 'off');
    set(handles.rightPB, 'Enable', 'off');
    set(handles.trainPB, 'Enable', 'off');
    set(handles.numPatternsText, 'String', '0');
else
    set(handles.numPatternsText, 'String', num2str(handles.column-1));
end
set(handles.accuracyText, 'Enable', 'off');
guidata(hObject, handles);

% --- Executes on button press in outputPB.
function outputPB_Callback(hObject, eventdata, handles)
try
    for R = 1:25
        val1 = get(handles.inputButtonHandles(R, 1), 'Value');
        D(R, 1) = val1;
    end
    if (get(handles.slpRB, 'Value') == 1)
        y = sim(handles.SLP, D);
    else
        y = sim(handles.MLP, D);
    end
    counter = 0;
    for R = 1:25
        val2 = round(y(R,1));
        if (val2 == 1)
            col = 'White';
        else
            col = 'Black';
        end
        set(handles.outputButtonHandles(R, 1), 'BackgroundColor', col);
        val3 = get(handles.targetButtonHandles(R, 1), 'Value');
        if (val2 == val3)
            counter = counter + 1;
        end
    end
end

```

```

        end
    end
    acc = (counter/25)*100;
    set(handles.accuracyText, 'String', strcat(num2str(acc), '%'));
    set(handles.accuracyText, 'Enable', 'on');
    guidata(hObject, handles);
catch
    errordlg('Neural network selected has not yet been
initialized.', 'Error');
end

% --- Executes on button press in leftPB.
function leftPB_Callback(hObject, eventdata, handles)
if (handles.columnSelector > 1)
    if ((handles.column == handles.columnSelector)&&(handles.column > 2))
        handles.columnSelector = handles.columnSelector - 2;
        handles.k = 0;
    else
        handles.columnSelector = handles.columnSelector - 1;
    end
    reflectToInput(handles);
    if (handles.columnSelector < handles.column - 1)
        set(handles.rightPB, 'Enable', 'on');
    end
    if (handles.columnSelector == 1)
        set(handles.leftPB, 'Enable', 'off');
    end
end

set(handles.deletePB, 'Enable', 'on');
guidata(hObject, handles);
% --- Executes on button press in rightPB.
function rightPB_Callback(hObject, eventdata, handles)
if (handles.columnSelector < handles.column - 1)
    handles.columnSelector = handles.columnSelector + 1;

    reflectToInput(handles);
    set(handles.leftPB, 'Enable', 'on');
    if (handles.columnSelector == handles.column - 1)
        set(handles.rightPB, 'Enable', 'off');
    end
end

set(handles.deletePB, 'Enable', 'on');
guidata(hObject, handles);

% --- Executes on button press in slpRB.
function slpRB_Callback(hObject, eventdata, handles)
set(handles.slpRB, 'Value', 1);
set(handles.mlpRB, 'Value', 0);

```

```

% --- Executes on button press in mlpRB.
function mlpRB_Callback(hObject, eventdata, handles)
set(handles.slpRB, 'Value', 0);
set(handles.mlpRB, 'Value', 1);

% --- Executes on button press in trainPB.
function trainPB_Callback(hObject, eventdata, handles)
P = handles.inputMatrix(1:25,1:handles.column-1);
T = handles.targetMatrix(1:25,1:handles.column-1);
if (get(handles.slpRB, 'Value') == 1)
    A(1:25,1) = zeros(25,1);
    A(1:25,2) = ones(25,1);
    handles.SLP = newp(A, 25);
    handles.SLP = train(handles.SLP,P,T);
else
    handles.MLP=newpr(P,T);
    handles.MLP.numlayers = 2;
    handles.MLP.layers{1}.size = 17;
    handles.MLP.layers{1}.transferFcn = 'logsig';
    handles.MLP.layers{2}.size = 25;
    handles.MLP.layers{2}.transferFcn = 'logsig';
    handles.MLP.layerConnect = [0 0;1 0];
    handles.MLP.outputConnect = [0 1];
    handles.MLP.biasConnect = [1;1];
    handles.MLP.trainFcn = 'trainrp';
    handles.MLP.divideParam.trainRatio = 100;
    handles.MLP.divideParam.testRatio = 0;
    handles.MLP.divideParam.valRatio = 0;
    handles.MLP.trainParam.epochs = 100;
    handles.MLP = train(handles.MLP,P,T);
end
set(handles.outputPB, 'Enable', 'on');
guidata(hObject,handles);

% -----
function aboutME_Callback(hObject, eventdata, handles)
About();

% -----
function exitME_Callback(hObject, eventdata, handles)
user_response = exitDlg('Title','Confirm Exit');

if (user_response == 'Yes')
    delete(handles.mainFig)
end

% --- Executes during object creation, after setting all properties.
function accuracyText_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function varargout = About(varargin)

```

```

% ABOUT M-file for About.fig
%     ABOUT, by itself, creates a new ABOUT or raises the existing
%     singleton*.
%
%     H = ABOUT returns the handle to a new ABOUT or the handle to
%     the existing singleton*.
%
%     ABOUT('CALLBACK',hObject,eventData,handles,...) calls the local
%     function named CALLBACK in ABOUT.M with the given input arguments.
%
%     ABOUT('Property','Value',...) creates a new ABOUT or raises the
%     existing singleton*. Starting from the left, property value pairs
are
%     applied to the GUI before About_OpeningFcn gets called. An
%     unrecognized property name or invalid value makes property
application
%     stop. All inputs are passed to About_OpeningFcn via varargin.
%
%     *See GUI Options on GUIDE's Tools menu. Choose "GUI allows only
one
%     instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help About

% Last Modified by GUIDE v2.5 06-Mar-2011 21:30:27

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                  'gui_Singleton',   gui_Singleton, ...
                  'gui_OpeningFcn', @About_OpeningFcn, ...
                  'gui_OutputFcn',  @About_OutputFcn, ...
                  'gui_LayoutFcn',  [], ...
                  'gui_Callback',    []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before About is made visible.
function About_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles     structure with handles and user data (see GUIDATA)
% varargin   command line arguments to About (see VARARGIN)

```

```

% Choose default command line output for About
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes About wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = About_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

function varargout = exitDlg(varargin)
% EXITDLG M-file for exitDlg.fig
%     EXITDLG by itself, creates a new EXITDLG or raises the
%     existing singleton*.
%
%     H = EXITDLG returns the handle to a new EXITDLG or the handle to
%     the existing singleton*.
%
%     EXITDLG('CALLBACK',hObject,eventData,handles,...) calls the local
%     function named CALLBACK in EXITDLG.M with the given input
%     arguments.
%
%     EXITDLG('Property','Value',...) creates a new EXITDLG or raises the
%     existing singleton*. Starting from the left, property value pairs
%     are
%     applied to the GUI before exitDlg_OpeningFcn gets called. An
%     unrecognized property name or invalid value makes property
%     application
%     stop. All inputs are passed to exitDlg_OpeningFcn via varargin.
%
%     *See GUI Options on GUIDE's Tools menu. Choose "GUI allows only
%     one
%     instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help exitDlg

% Last Modified by GUIDE v2.5 05-Mar-2011 09:43:41

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                  'gui_Singleton',  gui_Singleton, ...

```

```

        'gui_OpeningFcn', @exitDlg_OpeningFcn, ...
        'gui_OutputFcn', @exitDlg_OutputFcn, ...
        'gui_LayoutFcn', [] , ...
        'gui_Callback', []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before exitDlg is made visible.
function exitDlg_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles     structure with handles and user data (see GUIDATA)
% varargin   command line arguments to exitDlg (see VARARGIN)

% Choose default command line output for exitDlg
handles.output = 'Yes';

% Update handles structure
guidata(hObject, handles);

% Insert custom Title and Text if specified by the user
% Hint: when choosing keywords, be sure they are not easily confused
% with existing figure properties.  See the output of set(figure) for
% a list of figure properties.
if(nargin > 3)
    for index = 1:2:(nargin-3),
        if nargin-3==index, break, end
        switch lower(varargin{index})
            case 'title'
                set(hObject, 'Name', varargin{index+1});
            case 'string'
                set(handles.text1, 'String', varargin{index+1});
        end
    end
end

% Determine the position of the dialog - centered on the callback figure
% if available, else, centered on the screen
FigPos=get(0,'DefaultFigurePosition');
OldUnits = get(hObject, 'Units');
set(hObject, 'Units', 'pixels');
OldPos = get(hObject, 'Position');
FigWidth = OldPos(3);
FigHeight = OldPos(4);
if isempty(gcbf)
    ScreenUnits=get(0, 'Units');
    set(0, 'Units', 'pixels');
end

```

```

    ScreenSize=get(0,'ScreenSize');
    set(0,'Units',ScreenUnits);

    FigPos(1)=1/2*(ScreenSize(3)-FigWidth);
    FigPos(2)=2/3*(ScreenSize(4)-FigHeight);
else
    GCBFOldUnits = get(gcf,'Units');
    set(gcf,'Units','pixels');
    GCBFPos = get(gcf,'Position');
    set(gcf,'Units',GCBFOldUnits);
    FigPos(1:2) = [(GCBFPos(1) + GCBFPos(3) / 2) - FigWidth / 2, ...
                  (GCBFPos(2) + GCBFPos(4) / 2) - FigHeight / 2];
end
FigPos(3:4)=[FigWidth FigHeight];
set(hObject, 'Position', FigPos);
set(hObject, 'Units', OldUnits);

% Show a question icon from dialogicons.mat - variables questIconData
% and questIconMap
load dialogicons.mat

IconData=questIconData;
questIconMap(256,:) = get(handles.figure1, 'Color');
IconCMap=questIconMap;

Img=image(IconData, 'Parent', handles.axes1);
set(handles.figure1, 'Colormap', IconCMap);

set(handles.axes1, ...
    'Visible', 'off', ...
    'YDir'    , 'reverse' , ...
    'XLim'    , get(Img,'XData'), ...
    'YLim'    , get(Img,'YData') ...
    );

% Make the GUI modal
set(handles.figure1,'WindowStyle','modal')

% UIWAIT makes exitDlg wait for user response (see UIRESUME)
uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = exitDlg_OutputFcn(hObject, eventdata, handles)
% varargout cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

% The figure can be deleted now
delete(handles.figure1);

% --- Executes on button press in pushbutton1.

```



```

function pushbutton1_Callback(hObject, eventdata, handles)
% hObject      handle to pushbutton1 (see GCBO)
% eventdata    reserved - to be defined in a future version of MATLAB
% handles      structure with handles and user data (see GUIDATA)

handles.output = get(hObject, 'String');

% Update handles structure
guidata(hObject, handles);

% Use UIRESUME instead of delete because the OutputFcn needs
% to get the updated handles structure.
uiresume(handles.figure1);

% --- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)
% hObject      handle to pushbutton2 (see GCBO)
% eventdata    reserved - to be defined in a future version of MATLAB
% handles      structure with handles and user data (see GUIDATA)

handles.output = get(hObject, 'String');

% Update handles structure
guidata(hObject, handles);

% Use UIRESUME instead of delete because the OutputFcn needs
% to get the updated handles structure.
uiresume(handles.figure1);

% --- Executes when user attempts to close figure1.
function figure1_CloseRequestFcn(hObject, eventdata, handles)
% hObject      handle to figure1 (see GCBO)
% eventdata    reserved - to be defined in a future version of MATLAB
% handles      structure with handles and user data (see GUIDATA)

if isequal(get(hObject, 'waitstatus'), 'waiting')
    % The GUI is still in UIWAIT, us UIRESUME
    uiresume(hObject);
else
    % The GUI is no longer waiting, just close it
    delete(hObject);
end

% --- Executes on key press over figure1 with no controls selected.
function figure1_KeyPressFcn(hObject, eventdata, handles)
% hObject      handle to figure1 (see GCBO)
% eventdata    reserved - to be defined in a future version of MATLAB
% handles      structure with handles and user data (see GUIDATA)

% Check for "enter" or "escape"
if isequal(get(hObject, 'CurrentKey'), 'escape')
    % User said no by hitting escape

```

```
handles.output = 'No';

% Update handles structure
guidata(hObject, handles);

uiresume(handles.figure1);
end

if isequal(get(hObject, 'CurrentKey'), 'return')
    uiresume(handles.figure1);
end
```

APPENDIX B

FPGA LED Matrix Driver Schematic Diagram

