2012-01-01

Investigation of the Divcon Neuron to Increase the Performance of a Traditional Feed Forward Multi-Layer Perceptron and its Hardware Implementation

Jovan Saenz
University of Texas at El Paso, jsaenz@miners.utep.edu

INVESTIGATION OF THE DIVCON NEURON TO INCREASE THE PERFORMANCE OF A TRADITIONAL FEED FORWARD MULTI-LAYER PERCEPTRON AND ITS HARDWARE IMPLEMENTATION

JOVAN SAENZ

Department of Electrical and Computer Engineering

APPROVED:

_____
Patricia A. Nava, Ph.D., Chair

_____
Eric MacDonald, Ph.D.

_____
Virgilio Gonzalez, Ph.D.

_____
Stephen W. Stafford, Ph.D.

_____
Benjamin C. Flores, Ph.D.
Interim Dean of the Graduate School

To my wife, and Family for their Love and Support

INVESTIGATION OF THE DIVCON NEURON TO INCREASE THE PERFORMANCE OF A TRADITIONAL FEED FORWARD MULTI-LAYER PERCEPTRON AND ITS HARDWARE IMPLEMENTATION

by

JOVAN SAENZ

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

May 2012

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor and dissertation committee chair, Dr. Patricia Nava, who provided great support and guidance throughout the completion of this research. Without her support, I would not have been able to finish my doctorate degree studies. Her direction, patience, invaluable input, and effort in taking the time to review this dissertation during its different stages will never be forgotten.

I also would like to thank the UNIX lab staff for providing the resources necessary for the development and implementation of this research, which are partially supported by NSF Grant EIA-0325024 and CNS-0709438.

Finally, acknowledgement is also due to the faculty members that accepted to be part of my dissertation committee.

ABSTRACT

Artificial Neural Networks (ANNs) have been developed in an attempt to emulate the information processing capabilities of the biological brain. They offer an alternate computing approach to problems in which mathematical modeling is complicated, such as pattern recognition and pattern classification.

Since ANNs were proposed in the early 1940s, there has been a great amount of research effort dedicated to the development of new models that improve performance. Consequently, different architectures, a variety of activation functions, and distinct learning algorithms have been developed and implemented in different disciplines such as medicine, engineering, and science. In addition, ANNs have been combined with other alternate computing approaches such as *fuzzy logic*, *genetic algorithms*, and *quantum computing* to create hybrid systems in order to improve the performance of an ANN at the cost of making the system more complex. However, the majority of these efforts target the network level, and do not focus on the individual neuron.

This investigation focuses on the individual neuron and introduces the Divcon Neuron (DN) model, which increases the computing power of an ANN, when compared to the commonly used perceptrons. Two additional facets considered that derive from this research are first, the hardware implementation of the proposed model to observe the hardware resources utilized compared to perceptrons. The second aspect is the simulation time of the model to observe its computational benefits compared to the perceptron.

The DN proves to be an asset to performance in accuracy, simulation time, and efficiency.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 PROBLEM STATEMENT

Artificial Neural Networks (ANNs) offer an alternate computing approach to solve complex problems in which mathematical modeling is very complex such as pattern recognition and classification. They have been implemented to address problems in a variety of disciplines such as medicine, engineering, science, and business. ANNs have been applied in illness diagnosis & detection, speech recognition, fingerprint detection, vision systems, weather forecasting, stock market forecasting, and automation & control. In addition, since ANNs were first proposed in the early 1940s, there has been a great amount of investigation dedicated to the development of new models that improve their performance. As a result, different architectures, distinct activation functions, and a variety of learning algorithms have been developed and implemented for different applications. In addition, ANNs have been combined with other computing approaches such as fuzzy logic, genetic algorithms, and quantum computing, in an attempt to develop hybrid systems in order to obtain an increase in performance by taking advantage of the combined strengths of the computing approaches selected at the cost of increasing the complexity of the system in question. However, the majority of these efforts aims to improve the performance of the network as a whole and do not actually try to improve the processing power of the individual neuron. Consequently, there is a need to improve the processing power of the individual neuron in order to improve the overall performance of the ANN.

1.2 Proposed Hypothesis

Hypothesis statement: The performance of an Artificial Neural Network can be enhanced by increasing the computational power of the individual neurons which compose it. The hypothesis statement implies development of a new type of neuron, and this investigation introduces the *Divcon Neuron* (DN) model, which has higher processing power than the perceptron model. The DN model has the potential to increase the computing power of an ANN when compared to a network composed of standard perceptrons. In addition, another aspect that is to be considered is that this new model may contribute to a reduction in the number of neurons in the hidden layer, which possesses the potential to reduce the resources required for a hardware implementation of the network. This would be a great advantage since space is a critical restriction in hardware design. An additional aspect of consideration derived from this research, and observed in the experiments, is the simulation time of the proposed model, which is always good to consider when selecting a model for implementation of a particular application. The DN model is used to replace the perceptrons of a Feed Forward Multi-Layer Perceptron (MLP) using two approaches: *Option 1* replaces the perceptrons in the hidden layer of the MLP. Option 2 replaces the perceptrons in both the hidden and output layers of the MLP. Four different benchmark problems are used to evaluate the performance of the each of the two options described above, as well as the traditional MLP, which is used as the basis of comparison. For each of the options described previously, the number of DNs in the hidden layer is increased starting, arbitrarily, from five up to 40. The performance is to be evaluated for every DN added to the hidden layer until the number of 40 DNs is reached. In addition, the use of different activation functions is investigated to observe if a particular activation function contributes to an increase in the performance of the network. Finally, the traditional MLP and the MLP with DNs,

are implemented in hardware using a Hardware Description Language (HDL) to evaluate the potential savings in hardware resources obtained from the use of DNs.

1.3 Road Map

The description of this investigation is organized as follows: chapter 2 introduces the ANN background; chapter 3 describes the DN model, its implementation and the experiment results, chapter 4 discusses the hardware implementation of the DN model and the hardware resources utilized compared to the traditional perceptron, and finally, chapter 5 states conclusions drawn from testing and proposes future work.

Chapter 2

ARTIFICIAL NEURAL NETWORKS

2.1 Background

The interest in Artificial Neural Networks (ANNs) originates from the powerful information processing capabilities that they offer. ANNs were developed as an attempt to mimic the biological processes involved when the human brain processes information. Unlike traditional computing, which processes information sequentially, ANNs possess the inherent capability to process information in parallel. In addition, they can associate an output with a given input in a model-free environment, i.e. there is no need of a complex mathematical model a priori, since the mathematical model is iteratively obtained through a "training" process. Inspired by the outstanding processing power of the brain, ANNs try to emulate its processing capabilities; this is the reason why they have inherited the capabilities; for this reason, they have inherent capabilities to "learn" by examples and to be highly tolerant to failure due to their slow degradation.

The following sections discuss the fundamental theory of ANNs, how they are derived from their biological counterparts, and some applications in which ANNs have been used.

2.2 Biological Neural Networks

Biological neural networks consist of millions of interconnected biological neurons in the brain which, through complex electro-chemical reactions, are able to receive and transmit information using electrical signals [ROJ96]. The fundamental unit of these networks is the *"neuron,"* which is basically composed of three parts: the *cell body (*soma*)*, the *dendrites*

(terminals), and the *axon*, as depicted in Figure 2.1. The dendrites receive the input signals from other neurons and the axon transmits the output signal through its terminals (synapses).

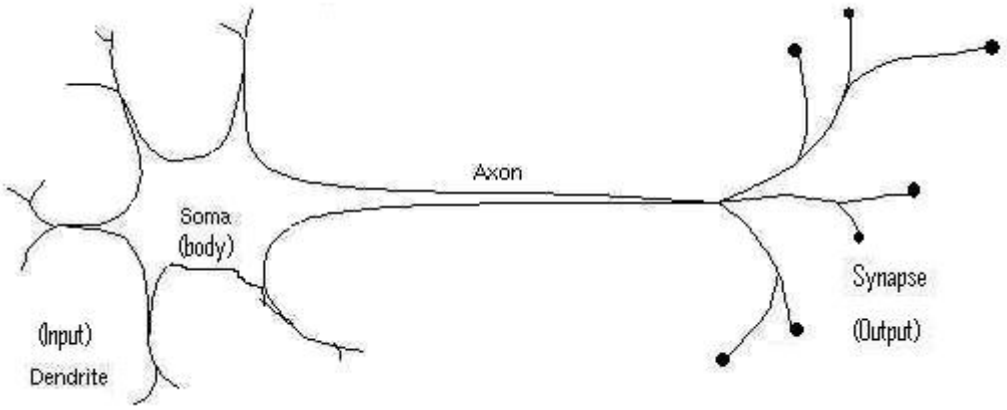


Figure 2.1 Biological Neuron

A biological neuron will only "fire," or send an output signal, if the overall intensity of the input signals received is higher than a certain threshold level. Otherwise, it will not send out any output signal. The signals that cause a neuron to transmit a signal are called *excitatory* (or positive) and the signals that prevent transmission are called *inhibitory* (or negative) [KAW00]. Consequently, if the excitatory signals outweigh the inhibitory signals by a level above a certain threshold, the neuron will be able to send an electrical signal through its axon to other neurons.

These millions of interconnected biological neurons possess incredible information processing capabilities (such as parallel processing) and fault tolerance, which inspired and gave birth to the development of artificial neural networks.

2.3 Artificial Neural Networks

ANNs, also known as parallel-distributed processors, connectionist models, adaptive systems, self-organizing systems, neurocomputing, and neuromorphic systems [BOS96, NEL90], were developed in order to emulate the biological neural network information processing capabilities. ANNs offer an alternate computing approach which has the potential to be used in applications difficult to be modeled with traditional computing such as pattern classification, clustering, and pattern recognition. ANNs are very good to approach problems that are difficult to model mathematically or problems where standard modeling would be become too computationally intensive and take too long to compute (such as facial recognition). In addition, ANNs offer the potential capability of parallel information processing, which is a great asset in intense-computing applications and would eliminate bottlenecks in sequential processing (traditional computing). Some of the areas in which ANNs have been implemented are robotics, vision systems, medicine, image processing, control, handwriting recognition, speech recognition, wafer alignment, air quality monitoring, data modeling, water networks modeling, forecasting, vehicle navigation, power amplifier modeling, target tracking, noise cancelling, sonar applications, fault analysis, physics, psychology, and business [ABD99, BOS96, BOU11, BUB11, CHA11, CHO07, DON10, FAU94, KAN11, KIM10, LEI11, LIK11, MEN10, MKA11, NEL90, POS09, QIN11, SAA11, SHA11, SHI09, SOR11, VOL11, and XIA11]. Unlike traditional computing, artificial neural networks learn by examples and can be trained (by presenting a set of given inputs with their corresponding outputs using an iterative process).

As described previously, ANNs were developed with the intent to create a system that would combine the strengths of the brain with the speed of machines. Their design, of course,

depends on the application, the number of neurons used, they way the neurons will be connected, the learning algorithm, and the quality of the input data as well as its quantity.

2.3.1 The Perceptron

The studies and development with regard to ANNs were proposed in the early 1940s by McCulloch & Pitts, but it was not until the early 1960s that the *perceptron* was introduced by Frank Rosenblatt [BOS96, CHO07, SAM07, and ZUR92] as an attempt to model the processing capability of a biological neuron. This particular model is one of the most commonly used in the literature and is the basic unit of the approach that is proposed in this research, which will be discussed in the following chapters. The perceptron model is shown in Figure 2.2.

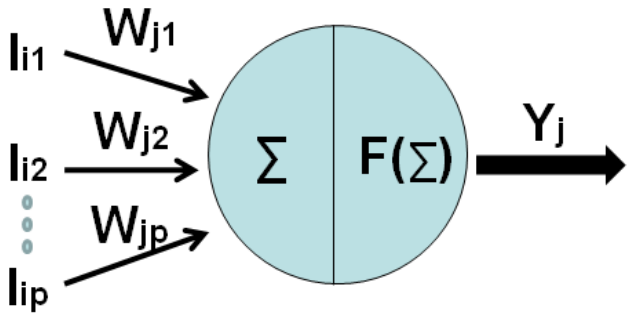


Figure 2.2 Perceptron Model

The inputs I1, I2, and I3 represent the dendrites of the biological neuron. The weights W1, W2, and W3 represent variable coefficients that determine the intensity of the input signals, which determine the output of the neuron (in the biological case, the intensity of the input signals determines whether the neuron transmits a pulse or not). The mathematical model of the perceptron is described as follows:

$$Y_j = f\left(\sum_{k=0}^{p} \left(x_{ik} * w_{jk} \right)\right), \tag{2.1}$$

Where p is the size of the input vector, x_{ik} is the k^{th} component of input vector (pattern) i, w_{jk} is the k^{th} weight coefficient corresponding to the link from input k to neuron j, $f(x)$ is an activation function, and Y_j is the output of neuron j.

As depicted in Figure 2.2, each neuron performs a weighted sum of its inputs, and the resulting accumulated sum is evaluated by an activation function, which provides the output of the perceptron.

The following section will discuss the concept of the activation function, which is used to generate the output of an artificial neuron.

2.3.2 Activation Functions

Activation functions "... are nonlinear, continuous functions that remain within some upper and lower bounds" [SAM07]. Activation functions are basically used to take advantage of their nonlinearity in order to make the processing of an ANN more powerful. There are several types of activation functions such as hyperbolic tangent, Gaussian, and sigmoid [FAU94, SAM07]. One of the most commonly used activation functions is the *sigmoid function*, which is shown in Figure 2.3.

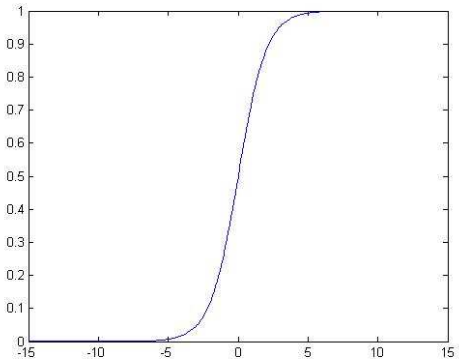


Figure 2.3 Sigmoid Function

The mathematical model of the unipolar sigmoid function is presented below:

$$\mathbf{y} = 1/ (1 + \exp (-x)), \tag{2.2}$$

This s-shaped activation function, commonly used in artificial neural networks, has the desired output range between 0 and 1. In addition, the nonlinear portion can be modified to cover a particular range of inputs. However, regardless of the type of activation function used, these functions are used to set the limits on the output data values. Moreover, mathematical proof on which activation function is best for the application at hand is not computationally feasible. Therefore, activation functions are usually selected from past experience, or by experimentation with the application in question to observe which performs best [CAL04].

2.3.3 Single Layer and Multilayer Networks

The basic architecture of ANNs is the *single layer network*, which is shown in Figure 2.4:

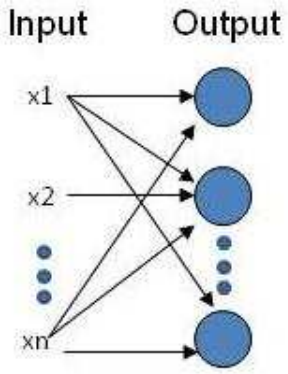


Figure 2.4 Single Layer Network

The single layer network consists of only one layer containing one or more neurons with several inputs connected to the network. This single layer is the actual *output layer*, which will

provide the response of the network. The way the inputs are connected to the output layer may vary depending on the type of ANN to be implemented.

The computing power of this network provides the capability to classify patterns that are *linearly separable*. A linearly separable problem is one that can classify different classes or groups with a single linear decision surface [ABD99]. A two-dimensional example is shown in Figure 2.5, in which *Classes 1 and 2* are separated by a linear decision surface, which in two dimensions is the line, *D1*.

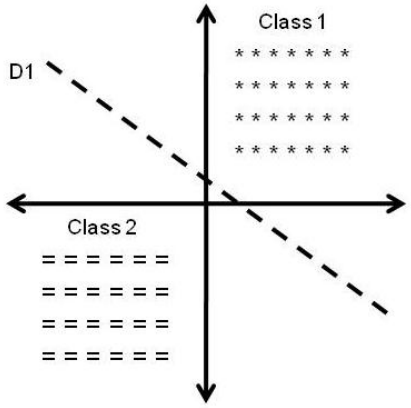
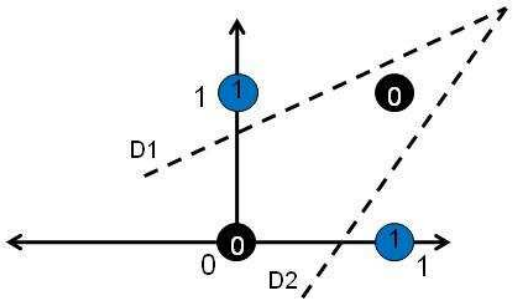


Figure 2.5 Linearly Separable Patterns

However, a disadvantage of the single layer network is that it is not capable of classifying *linearly inseparable* patterns, which cannot be classified with a single linear surface. A classic problem that shows linear inseparability is the XOR logic function shown in Figure 2.6.



X1	X2	Z
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.6 Linearly Inseparable XOR Logic Function

The XOR cannot be classified with only one linear decision surface, or line, as shown in Figure 2.6; the XOR needs two decision lines (D1 and D2) to classify the outputs correctly. This type of problems gave rise to the development of the *multilayer network*. The multilayer network basically consists of the introduction of additional layers of neurons cascaded between the output layer and the inputs. Figure 2.7 shows the general structure of a multilayer network, which provides greater computational capabilities and is able to solve linearly inseparable problems.

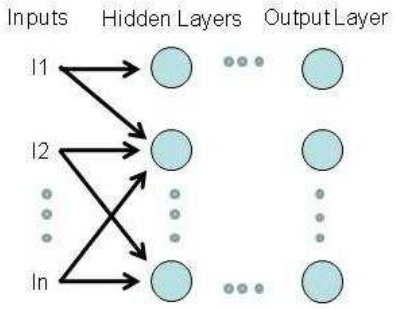


Figure 2.7 Multilayer Network

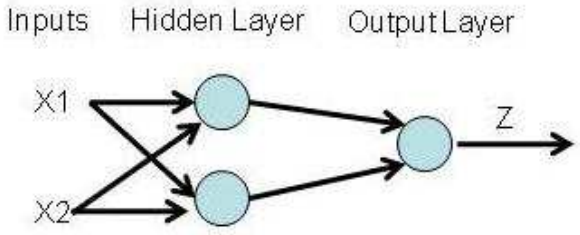


Figure 2.8 Network architecture that solves XOR problem

The cascaded layers of neurons between the inputs and the output layer in a multilayer network are referred to as *hidden layers*. Figure 2.8 shows an example of a multilayer network that can solve the XOR pattern classification problem successfully [FAU94, KAW00]. This configuration contains the two binary inputs, one hidden layer containing two hidden neurons

and one neuron in the output layer providing the expected binary output (0 or 1). In the XOR function case, there are only two classes, and two hidden neurons are enough to perform the classification successfully. Depending on the particular application in question, a multilayer network can have more than one hidden layer. However, most applications only use one hidden layer. The number of neurons in the hidden layer (called hidden neurons), for a specific application can be calculated. However, it is usually very difficult to calculate, and therefore, is usually determined from experience or by trial and error. It is up to the designer to make these decisions. When designing an ANN, the *network architecture*, the *activation function*, and the *learning algorithm* have to be selected. The following section will discuss how an ANN actually "learns" using what is referred to as a *learning algorithm*.

2.3.4 Learning Process

One of the most important characteristics of ANNs is their capability to "learn". Learning is basically the use of an algorithm (referred to as learning algorithm) to adjust the weights through an iterative process in order to find the set of weight values that will allow the artificial neural network to classify a given set of patterns successfully. There are many different learning algorithms and selection of the appropriate algorithm is based on the architecture and connectivity of the neurons, such as: *gradient descent*, in which weights are modified by an amount proportional to the first derivative of the error with respect to the weight; *backpropagation*, which is derived from the gradient descent rule; *competitive learning*, in which only one neuron is selected to provide the output and only the weights connected to that neuron are updated; and *r-propagation*, in which weights are updated based on the sign of the gradient; [BOS96, CHO07, FAU94, KAR96, NEL90, SAM07, and ZUR92]. There are two types of

learning: *supervised* and *unsupervised*. *Supervised learning* involves feeding the ANN with a set of input patterns and their associated outputs. Depending on the learning algorithm used, the weights are adjusted based on an error calculation, related to the actual output generated by the network and the target output presented, until an acceptable error value is reached (gradient descent is an example of supervised learning) [CHO07, KAR96, and SAM07]. On the other hand, *unsupervised learning* is related to introducing the set of input patterns to the network without providing the target outputs. In this case, the network organizes the data into groups or clusters based on the similarities shared by the features of the input patterns (competitive learning is an example of unsupervised learning) [CHO07, KAR96, and SAM07]. The following section describes the particular learning algorithm that is used in this research, which is one of the most commonly used learning techniques known as *backpropagation learning*.

2.4 Backpropagation Learning

The backpropagation algorithm is relatively easy to implement and is frequently used in many applications such as speech recognition, wafer alignment, virtual simulation, license plate recognition, and forecasting, [LI02, LI11, KIM10, YAN11, and YU09]. Backpropagation looks for the minimum output error in the associated weight space using a gradient based method [BOS96, CHO07, FAU94, KAW00, and ROJ96]. This algorithm is by and far the most commonly used in one of the most popular network architectures: the *multilayered perceptron*, which is as its name indicates, a multilayer network containing perceptrons that are interconnected. This is the network architecture used in this research and will be discussed in the following chapters. The name "backpropagation" comes from the way in which the weights of an ANN are adjusted. When a neural network receives an input pattern, it "propagates" through

the network and generates an output. The result is then compared with the target output associated with the input pattern in question in order to calculate an error. This error is then backpropagated through the network, that is, from the output layer all the way to the inputs, and all the weights are adjusted based on this error calculation. This process is repeated until a minimum error value is reached or until a predefined number of iterations occur. The algorithm for backpropagation learning can be described in more detail (assuming the use of perceptrons, a unipolar sigmoid activation function, and one hidden layer) as follows:

1) An Input pattern is propagated through the ANN all the way to the output layer as follows:

- *Input connected to hidden layer:* Each input is multiplied by its corresponding weight and each of these products is added resulting in a summation of products, which is evaluated by an activation function.

$$S = \sum_{k=0}^{p} (x_{ik} * w_{jk}) \tag{2.3}$$

$$\mathbf{Y_{hj}} = 1/ (1 + \exp (-S)) \tag{2.4}$$

Where p is the size of the input vector, x_{ik} is the k^{th} component of input vector (pattern) i, w_{jk} is the k^{th} weight coefficient corresponding to the link from input k to neuron j, S is the accumulated sum, and Y_{hj} is the sigmoid activation function that generates the output of hidden neuron j.

- The *next layer* is the output layer. In this case the outputs of the hidden layer are used as inputs. Once the output layer generates a result, the output is compared to the target output and an error is calculated.

2) The calculated error is propagated through the network backwards as follows:

- Calculate the error δ_{jo} for every neuron j in the output layer:

$$\boldsymbol{\delta_{jo}} = (t_{ij} - Y_j) (1 - Y_j) (Y_{hj}) \tag{2.5}$$

Where t_{ij} is the target output for neuron j associated with input pattern i, and

Y_j is the output generated by output neuron j.

- Calculate the delta weight value ΔW_{jk} for every weight k connected to output neuron j:

$$\mathbf{\Delta W_{jk}} = \alpha\ (\delta_{jo})\ (Y_{kj}) + (m * \Delta W_{jk}) \tag{2.6}$$

Where α is the *learning rate*, which is used to determine how fast an ANN learns, Y_{kj} is the output of neuron k in the hidden layer connected to output neuron j, and m is a parameter named *momentum*, which will be discussed in the next section.

- Calculate the error δ_{kh} for every neuron k in the hidden layer:

$$\boldsymbol{\delta_{kh}} = (1 - Y_{hk})\ (Y_{hk})\ (\sum \delta_{jo} * W_{jk}) \tag{2.7}$$

Where W_{jk} is the weight k connected to output neuron j, and $Y_{\boldsymbol{hk}}$ is the output of hidden neuron k.

- Calculate the delta weight value ΔW_{kh} for every weight h of hidden neuron k:

$$\mathbf{\Delta W_{kh}} = \alpha\ (\delta_{kh})\ (X_{ik}) + (m * \Delta W_{kh}) \tag{2.8}$$

3) Calculation of new weights:

- For weight k connected to output layer neuron j (output layer weights):

$$\boldsymbol{W_{jk}} = W_{jk} + \Delta W_{jk} \tag{2.9}$$

- To update the weights for the hidden layer:

$$\boldsymbol{W_{kh}} = W_{kh} + \Delta W_{kh} \tag{2.10}$$

This process is repeated until the error is minimized to the particular tolerance specified.

The following section will discuss the learning rate and momentum parameters introduced in the backpropagation algorithm.

2.4.1 Learning Rate and Momentum

The *learning rate* is used to determine how fast an ANN learns and is generally a value between 0 and 1. In other words, it is the step size used by the network to move in the weight space in order to adjust the weights and find the right set of weight values that will minimize the classification error and produce the desired results [SAM07]. If the learning rate value is very small, it will take a long time for the network to learn, that is, the step size used to reach the desired set of weights is very small and is shown in Figure 2.9. On the other hand, if the learning rate is very high, the network can learn faster. However, a high learning rate can cause the network to miss the target set of weights completely and the network will not be able to learn, as shown in Figure 2.10.

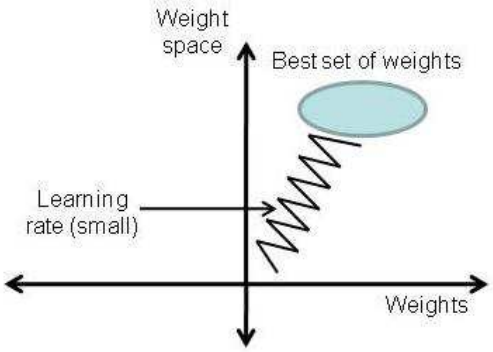


Figure 2.9 Small Learning Rate

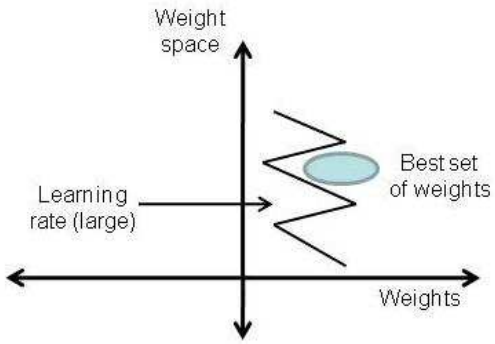


Figure 2.10 Large Learning Rate

In backpropagation, there is a problem that causes a neural network not to find the right set of weights to arrive to the desired solution. Backpropagation looks for the *minimum error* or *global minimum*, however, sometimes the network may fall into what is referred to as *local minimum*. As shown in Figure 2.11, a local minimum is a form of trap, which can prevent the neural network from finding the global minimum resulting in oscillations [CHO07].

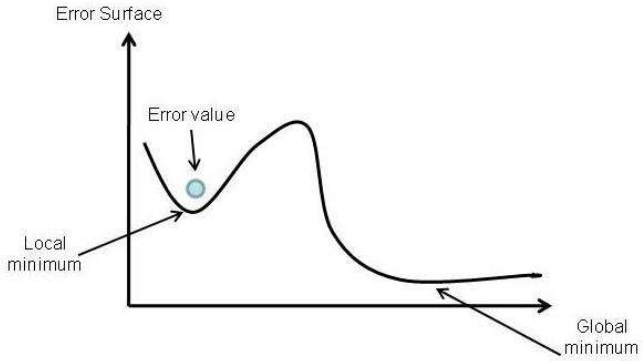


Figure 2.11 Local Minimum

One of the methods developed to attempt to deal with the local minimum problem is the incorporation of the *momentum* parameter. Momentum can have a value between 0 and 1. In addition, momentum can filter-out high frequency variations of the error surface [CHO07]. In other words, it increases the change in weight values until it allows the network to be able to jump out of the trap represented by the local minimum.

2.5 Artificial Neural Network Architectures

There are two main categories of architectures of ANNs which are chosen depending on the application in question, and this section will go over the most commonly implemented. One of the most commonly used for classification and pattern recognition problems is the *Feed Forward Network* (FFN). As its name implies, this is a multilayer network made up of neurons connected as the structure shown in Figure 2.12; in fact, the network that solves the XOR problem shown in Figure 2.8 is an example of a FFN:

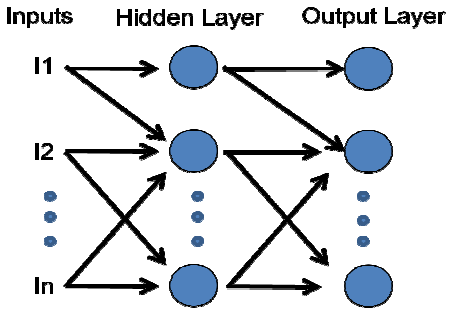


Figure 2.12 Feed Forward Network

In this architecture, there are no feedback connections of any kind. In addition, the information is processed *forward*, that is, from the input layer to the output layer and the number of neurons used in each layer and the number of layers depends on the application [CHO07, KAR96]. The other category is known as *Recurrent Neural Networks* (RNNs), and uses an architecture that contains one or more feedback connections as shown in Figure 2.13. These feedback connections can be used to introduce memory to the network and make it appropriate for their use in applications related to prediction and time-series problems [CAP11, CHO07, MAN01, and SAM07].

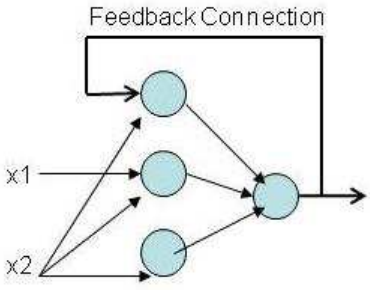


Figure 2.13 Recurrent ANN

There is another architecture approach named *Modular Neural Networks* (MNNs) [GRA09, LIY08, MAR08, and SIL08], in which the network is made up of small ANNs as shown in Figure 2.14:

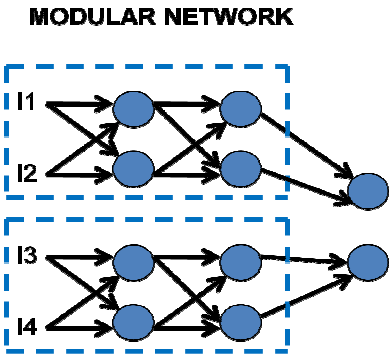


Figure 2.14 Modular Neural Networks

In this architecture, each network or *module* receives only a portion of the input data with the purpose of having each particular module become an "expert" in recognizing specific features of the input data. The architecture of each module can be a FNN, a RNN or a combination of both. The architecture selection is a key factor in an ANN's implementation. Based on these fundamental architectures, many different types of ANNs have been developed such as *Probabilistic Neural Networks* (PNNs), which are FFNs that use Gaussian activation functions, do not require iterative training, and only need one pass of all the training patterns in order to train the network [JIE09, SON07, SPE92, TIA09, TRI10, YOU08, and ZAK97]. Another type of ANNs is the *Stochastic Neural Network (SNN)*, which are RNNs that introduce random variations into the network usually through the weights or the activation functions [BRO01, GER08, JEN01, ONO09, SAN02, and ZHA05]. In addition, another type of ANNs is the *Spiking Neural Network* (SNN), which are RNNs that attempt to model biological neural behaviour more closely which allows taking into account the timing of inputs and I/O is represented as a series of spikes [ALL09, HAS09, LIU09, PIT09, TOR09, and WUQ09].

2.6 Quantum Neural Networks

Quantum neural networks (QNNs) derive from the combination of quantum computing and ANNs. QNNs have been applied in different applications such as speech recognition, control, signal prediction, speech enhancement, stochastic logic, signal recognition, circuit simulation, and letter recognition [CAO09, LEE04, LI02, LI03, LI04, LI05, LI09, LIN04, ONO09, TSA05, UDR05, and ZHA06]. In addition, quantum computing has been applied in this research to contribute to the development of the DN model. The following section will introduce the basic theory behind quantum computing and how it has been combined to ANNs.

2.6.1 Quantum Computing Background

Quantum computing is derived from quantum mechanics theory, which is used to analyze light wave polarization theory. Polarization of a light wave is the direction in which the **E** field of the wave oscillates up/down or left/right [MOR08]. Polarization can be horizontal or vertical, and if two polarized light waves are added as shown in Figure 2.15, a diagonally polarized light wave can be obtained:

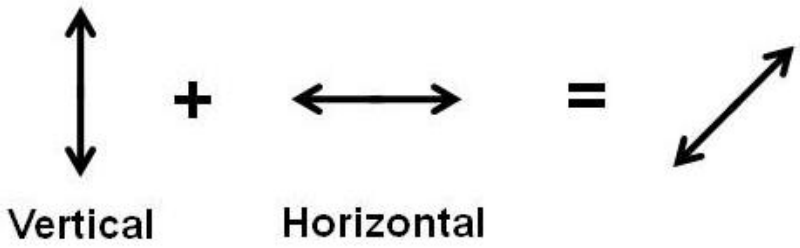


Figure 2.15 Beam of light diagonally polarized with intensity = 1

The above diagonal light wave can be represented as a *unit vector* having components *a* (horizontal) and *b* (vertical) [MOR08], as depicted in Figure 2.16 below:

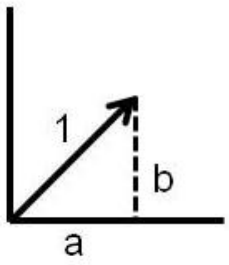


Figure 2.16 Unit vector representation of a light wave

Considering Figure 2.16, and using the Pythagorean Theorem, each component can be calculated as described below (assuming a 45 degree angle):

$$a = \cos(45) = (1/\sqrt{2}) \tag{2.11}$$

$$b = \sin(45) = (1/\sqrt{2}) \tag{2.12}$$

$$(1/\sqrt{2})^2 + (1/\sqrt{2})^2 = 1 \tag{2.13}$$

Therefore, a photon of light can be represented as a vector in Dirac notation [KAY07, and MCM08] as follows:

$$|photon\rangle = a|X\rangle + b|Y\rangle \tag{2.14}$$

Analogous to the representation of a photon, quantum computing derives its basic unit of information called a *qbit* [CHE07, KAY07, MCM08, and MOR08]. A qbit is represented as a vector in Dirac notation as follows:

$$|Qbit\rangle = a|0\rangle + b|1\rangle \tag{2.15}$$

Similar to traditional binary computing, a qbit can be in two states: $|0\rangle$ (zero) and $|1\rangle$ (one), where a^2 and b^2 are the probabilities of being in state $|0\rangle$ and $|1\rangle$ respectively. Quantum states can be represented by matrices as well. For example, quantum state $|0\rangle$ can be represented as $(1\ 0)^T$ and quantum state $|1\rangle$ can be represented as $(0\ 1)^T$.

Quantum computing has been combined with ANNs to improve the performance compared to the crisp ANN, that is, a crisp ANN is a network that has not been combined with a

different computing approach. The use of quantum neural networks, which is the result of this combination, is applied in this research focused in the development of the DN model.

Chapter 3

IMPLEMENTATION OF DIVCON NEURONS

3.1 Background

In General, previous work in the literature has been dedicated to the improvement of ANNs by using a variety of activation functions [CAL04, CHO10, LEE04, LIL10, LIU10, NIE11, POS09, SON07, WUJ09, XU07], distinct learning algorithms [DEL10, FAU94, KAR96, LUD10, PAP11, RAJ10, SAM07, SHA11, XIA10], and different architectures [AHM04, HUN11, JIA07, LIY08, MAR08, MIT02, MIS08, SIL08, VIL11]. In addition, ANNs have been combined with other alternate computing approaches such as Fuzzy Logic [GUP11, JAN93, HEL08, LIU08, MAD07, PED06, SAE03, STA07, XIA06, YIL11], Genetic Algorithms [FAY10, HIG06, HU10, KON04, LIU10, LOP08, NED05, SAS09, and YU09] and Quantum Computing [AMB05, CHE07, KAY07, LEE04, LIF02, LIF03, LIF04, LIF05, LIN04, MCM08, MOR08, TSA05, ZHA04, and ZHA06] to create hybrid systems that benefit from the advantages of the computing approaches selected at the cost of making the system more complex. However, all this work attempts to increase the performance at the network level. In other words, these efforts do not concentrate on the individual neuron. At the neuron level, several models dedicated to increase its processing power have been developed and implemented such as quantum neuron models [CAO09, LIF09, MOR06, XIA09, and ZHA06], pi-sigma models [SHI97 and YAD03], generalized neuron model [NAR07], second order neuron [HOM02], higher order neuron [MIN02], and other models [CHA05, EFE09, HIK03, HIS11, KOT08, PAN11, SHI10, SIN01, TRI11, WAN05, WAN08, XU08, YAD04]. However, most of the work in the literature targets the network level when attempting to improve performance.

3.2 Initial Divcon Neuron Model

The main objective of this research is to test the hypothesis: The performance of an ANN can be enhanced by increasing the computational power of the individual neurons which compose it. To test the hypothesis, the concept of the divcon (diverge / converge) neuron model is introduced, and its potential to increase the performance of a Feed Forward MLP is investigated. In addition, two more facets are considered as well: The first is the resources required to implement the DN model in hardware in comparison to the hardware resources utilized by the traditional perceptron model; The second aspect that is considered is the simulation time of the network using DNs, to observe the computation load provided by the DNs, which is something to consider when a neuron model is selected. The initial approach to this neuron model was designed as shown in Figure 3.1:

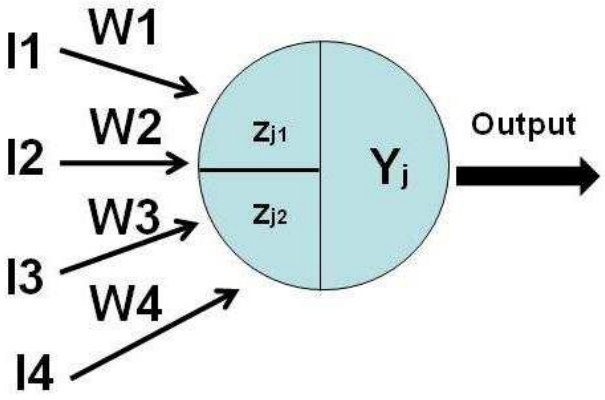


Figure 3.1 Divcon Neuron Model

Given an input vector pattern of length p, the mathematical model is shown below:

$$z_{j1} = f(\sum_{k=0}^{(\frac{p}{2})-1}(x_{ik} * w_{jk})) \tag{3.1}$$

$$z_{j2} = f(\sum_{k=p/2}^{p}(x_{ik} * x_{jk})) \tag{3.2}$$

$$Y_j = f((z_{j1} * v_{j1}) + (z_{j2} * v_{j2}) + v_{j3}) \tag{3.3}$$

Where x_{ik} is the k^{th} component of input vector (pattern) i, w_{jk} is the k^{th} weight coefficient corresponding to the link from input k to neuron j, $f(x)$ is the activation function, z_{j1} is the output of the first half of inputs evaluated, z_{j2} is the output of the remaining half of the inputs evaluated, v_{j1} and v_{j2} are internal weights of neuron j, v_{j3} is a bias term of neuron j, and Y_j is the output of neuron j.

The DN operates as follows:

1) All inputs connected to the neuron are *diverged* into two groups.
2) Each group goes through a nonlinear mapping separately (z_{j1} and z_{j2} respectively).
3) The results z_{j1} and z_{j2} are *converged* by multiplying them by the internal weights v_{j1} and v_{j2} respectively. These products and a bias term are summed and finally, the result goes through an additional nonlinear mapping, resulting in the neuron's output Y_j.

The divcon neuron has higher processing power than the commonly used perceptron, as evidenced by the fact that a single divcon neuron can successfully compute the XOR logic function. A single traditional neuron (perceptron) does not have the capability to successfully compute the XOR logic function. In order to compute the XOR function using perceptrons, a multilayer perceptron is needed. The additional nonlinear mapping capability of the divcon neuron (i.e. the use of two internal activation functions), provides a level of complexity to the divcon neuron that is characterized by higher processing power, when compared to the traditional perceptron model.

3.3 Implementation of the Divcon Neuron

The first step is the selection of a traditional Feed Forward MLP as the "base case", to use as the basis of comparison. The selected network has been used in different research projects [CAL04, NAV96, NAV98, SAE03] within the neuro-fuzzy systems research group at the ECE department of The University of Texas at El Paso, and its features are listed below:

- MLP with only one hidden layer and 22 neurons in the hidden layer
- Learning Algorithm used: Backpropagation with momentum
- Implemented in ANSI C and C ++ [BLU92, DEW03, MAS93, and ROG97] language under Linux environment

The performance of the traditional MLP is evaluated and compared to an identical MLP that uses divcon neurons instead of perceptrons. Four different benchmark problems are selected to evaluate the performance of the networks and the following section goes over each application problem in more detail.

3.3.1 Benchmark Problems

The four benchmark problems used for this investigation were selected from the UCI Repository of Machine Learning Databases [UCI10] and are described below:

1. Wine Dataset, used in other projects such as [LEI09, TAT10, and XIA06]:
    - 178 training patterns and 74 testing patterns
    - Contains 3 different classes, that is, 3 types of wines
    - Input vector length = 13 elements, output neurons required = 2
2. Vowel Dataset, used in other projects such as [CAL04, NAV96, and SAE03]:
    - 528 training patterns and 462 testing patterns
    - Contains 11 classes, that is, 11 different vowel sounds in the English language

- Input vector length = 10 elements, output neurons required = 4

3. Yeast Dataset, used in other projects such as [HOH04, KRA07, and RAV07]:
    - 890 training patterns and 594 testing patterns
    - Contains 10 classes, that is, 10 different protein localization sites
    - Input vector length = 8 elements, output neurons required = 4
4. Letter Dataset, used in other projects such as [INO05, ISL08, and KUM99]:
    - 16000 training patterns and 4000 testing patterns
    - Contains 26 classes, that is, the 26 capital letters of the English alphabet
    - Input vector length = 16 elements, output neurons required = 5

3.3.2 Implementation Approach

This investigation evaluates two distinct approaches in which the DNs are implemented. In option 1, the performance of the traditional network is evaluated and compared against the same network in which the DNs are used to replace the perceptrons in the hidden layer, as depicted in Figure 3.2 below:

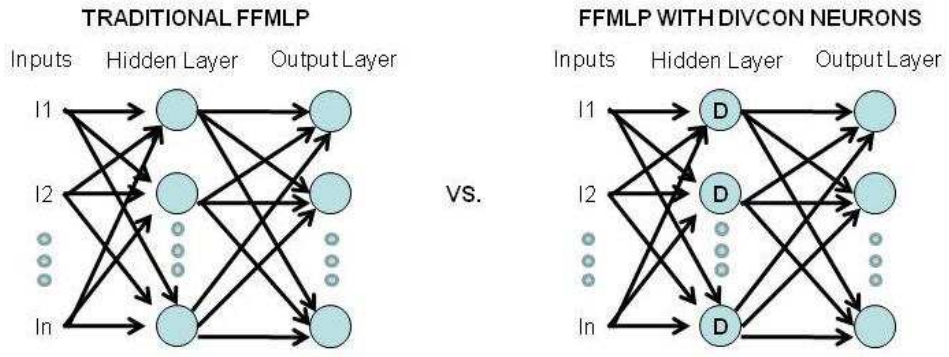


Figure 3.2 Divcon Neurons in Hidden Layer

The performance of the MLP containing DNs is evaluated to observe if the performance of the network is increased and if the number of neurons in the hidden layer can be reduced, therefore:

- Divcon neurons are added in the hidden layer starting with 5 up to 40 divcon neurons in the hidden layer. The performance of the network will be evaluated every time an additional divcon neuron is added until the number of 40 divcon neurons is reached.
- The best performance of the network with the minimum number of divcon neurons will be selected and will be compared to the performance of the traditional network.
- The same process is repeated for each benchmark problem.

For option 2, the same procedure is followed, the performance of the traditional network is evaluated and compared against the same network using DNs in all layers of the network, that is, all perceptrons in both the hidden the output layer are replaced with DNs as depicted in Figure 3.3 below:

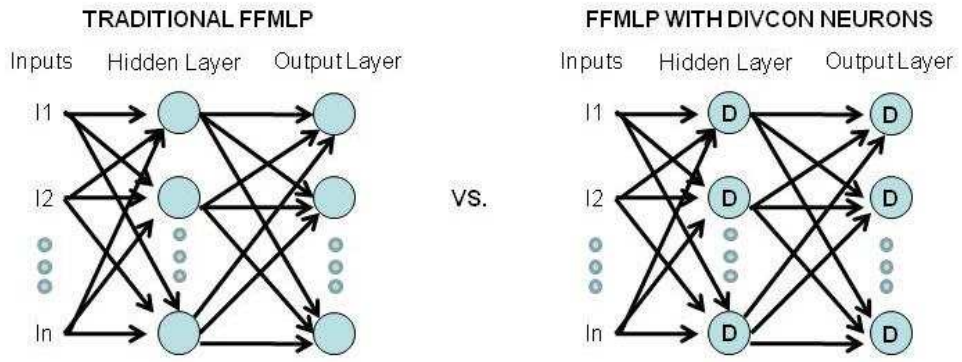


Figure 3.3 Divcon Neurons in All Layers

3.3.3 Performance Evaluation

In order to evaluate the performance of the MLP, the value of two internal parameters (learning rate and momentum) is modified in order to observe at which particular value of the

parameters in question the best performance of the network is achieved. The procedure is described as follows:

- The learning rate and momentum are initialized to a value of *0.1*
- The MLP is trained and tested and the performance is recorded in an output file.
- The momentum value is then increased by a step size of 0.1 until a maximum value of 0.9 is reached, and every time the momentum is increased, the performance of the network is evaluated and recorded in the output file.
- Once the momentum reached its maximum value, the learning rate is increased using the same step size of 0.1 and the same procedure described previously is repeated until the learning rate reaches a maximum value of 0.9.

The performance of the network is evaluated by selecting the best result provided by a particular value of the two internal parameters. In addition, the lowest performance and the average of all results are recorded as well.

Table 3.1 shows the performance of the traditional MLP using the binary sigmoid activation function for all possible values of the learning rate and momentum using the wine benchmark problem. As observed on the table, the best performance of the traditional network for the wine benchmark problem is **100%** recognition. That is, all the testing patterns were recognized successfully. The average recognition was **94.35%**, which is basically the average of all the results recorded. Finally, the worst performance was **0%** recognition, that is, none of the testing patterns were recognized successfully.

Table 3.1 Traditional MLP wine benchmark problem results

CRISP ARTIFICIAL NEURAL NETWORK
WINE TESTING DATA SIMULATION RESULTS

	LEARNING RATE								
M	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	100%	100%	100%	100%	100%	100%	100%	100%	100%
0.2	100%	100%	100%	100%	100%	100%	100%	100%	100%
0.3	100%	100%	100%	100%	100%	100%	100%	100%	100%
0.4	100%	100%	100%	100%	100%	100%	100%	100%	100%
0.5	100%	**100%**	100%	100%	100%	100%	100%	100%	100%
0.6	100%	100%	100%	100%	100%	100%	100%	100%	40.5%
0.7	100%	100%	100%	100%	100%	100%	40.5%	40.5%	100%
0.8	100%	100%	100%	40.5%	100%	100%	40.5%	100%	**0%**
0.9	100%	40.5%	100%	100%	100%	100%	100%	100%	100%

M = Momentum

Best Classification = 100%
Lowest classification = 0%
Average classification = 94.35 %

Type: Feed forward Multi-layer Perceptron
Learning Algorithm: Back propagation
 # hidden layers = 1
 # Inputs = 13
 # Output neurons = 2
 # hidden neurons = 22

3.4 Simulation Results for Phase 1 Model

The performance of the networks is evaluated using different activation functions to observe if a particular activation function contributes to better recognition performance.

Extensive experimental results found of this investigation are reported as follows, for each distinct activation function tested:

- Traditional network performance for each benchmark problem;
- Option 1 (DNs utilized in the hidden layer only) performance for each benchmark problem;
- Best performance (selected with the minimum number of divcon neurons in the hidden layer) compared against the traditional network;

- Option 2 (network composed entirely of DNs) performance for each benchmark problem;
- Best performance (selected with the minimum number of divcon neurons in the hidden layer) will be compared against the traditional network.

3.4.1 Results using a Binary Sigmoid Activation Function

Using the *binary sigmoid function* depicted in Figure 2.3, as the activation function, Table 3.2 shows the best performance of the traditional MLP for all benchmark problems (accuracy measured in percentage of correct classifications):

Table 3.2 Traditional MLP Performance using Binary Sigmoid Function

Dataset	Max	Min	Average
Wine	100%	0%	94.35%
Vowel	68.18%	9.1%	47.74%
Yeast	75.08%	30.05%	69.83%
Letter	61.62%	3.9%	15.94%

Option 1 (DNs in the hidden layer only), utilizing a binary sigmoid activation function, is implemented, and the Wine dataset is tested, as illustrated in Table 3.3:

Table 3.3 Option 1 Performance for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	0%	76.21%	**14**	100%	0%	74.74%	**30**	100%	0%	75.24%
6	100%	0%	76.71%	**16**	100%	0%	74.39%	**32**	100%	0%	73.77%
7	100%	0%	77.34%	**18**	100%	0%	74.51%	**34**	100%	0%	78.18%
8	100%	0%	76.71%	**20**	100%	0%	74.51%	**36**	100%	0%	73.77%
9	100%	0%	76.71%	**22**	100%	0%	74.11%	**38**	100%	0%	71.57%
10	100%	0%	76.71%	**24**	100%	0%	78.18%	**40**	100%	0%	73.04%
11	100%	0%	75.88%	**26**	100%	0%	73.74%				
12	100%	0%	75.98%	**28**	100%	0%	73.77%				

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network, as shown in table 3.4 below:

Table 3.4 Option 1 vs. Traditional Network for Wine Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0%	94.35%
Approach 1	5 DNs	100%	0%	76.21%
Traditional	5	40.54%	0%	26.53%

As table 3.4 indicates, both the traditional FFMLP and Option 1 provide a perfect recognition performance of **100%**. However, it takes 22 perceptrons in the traditional FFMLP to achieve this performance. On the other hand, only *5 divcon neurons* are needed to provide the same performance. If the number of perceptrons in the traditional network is reduced to match the number of divcon neurons selected, that is, 5 perceptrons in the hidden layer, the performance drops to 40.54%.

Table 3.5 shows the results of Option 1 for the Vowel dataset:

Table 3.5 Option 1 Results (binary sigmoid) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	30.74%	9.1%	14.4%	**10**	38.53%	9.52%	17.59%	**24**	47.84%	12.77%	21.5%
6	35.71%	9.52%	15.06%	**11**	40.04%	9.74%	17.17%	**28**	44.59%	15.58%	26.97%
7	30.3%	9.1%	14.79%	**12**	34.63%	11%	17.87%	**32**	45.02%	14.29%	28.48%
8	50.43%	9.52%	15.95%	**16**	41.34%	11.26%	19.82%	**36**	44.16%	17.32%	30.35%
9	31.39%	9.1%	15.38%	**20**	47.84%	12.77%	21.5%	**40**	69.26%	17.75%	31.58%

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.6 below:

Table 3.6 Option 1 vs. Traditional (binary sigmoid) Network for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	68.18%	9.1%	47.74%
Approach 1	8 DNs	50.43%	9.52%	15.95%
Traditional	8	40.54%	0%	26.53%

The Vowel dataset is a very difficult dataset, as is well documented in the literature, as well as the ML Repository documentation [UCI10]. As table 3.6 indicates, the traditional network performed better than the divcon neurons with recognition of **68.18%**. The best result selected with the minimum number of divcon neurons (8 in this case) provided a recognition performance of **50.43%**. However, it is important to point out that when the number of perceptrons in the traditional network matches the number of divcon neurons selected, the performance of the traditional network falls to **40.54%**. Thus, the Divcon provides a more efficient system for the given set of processing elements.

Table 3.7 shows the results of option 1 for the yeast dataset:

Table 3.7 Option 1 Results (binary sigmoid) for the Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	68.69%	39.1%	45.72%	**10**	68.69%	39.1%	48.67%	**24**	68.69%	39.1%	50.7%
6	68.69%	39.1%	45.94%	**11**	68.69%	39.1%	48.49%	**28**	68.69%	39.1%	48.91%
7	88.22%	39.1%	47.19%	**12**	68.69%	39.1%	50.25%	**32**	88.22%	39.1%	53.32%
8	68.69%	39.1%	47.13%	**16**	68.69%	39.1%	49.27%	**36**	68.69%	39.1%	52%
9	68.69%	39.1%	46.93%	**20**	68.69%	39.1%	48.99%	**40**	88.22%	39.1%	52.92%

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.8 below:

Table 3.8 Option 1 vs. Traditional (binary sigmoid) for Yeast Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	75.08%	30.05%	69.83%
Approach 1	7 DNs	88.22%	39.1%	47.19%
Traditional	7	39.1%	39.1%	39.1%

As table 3.8 indicates for the Yeast data, the performance of the traditional network of **75.08%** was surpassed by the use of divcon neurons with a performance accuracy of **88.22%**. In addition, only 7 divcon neurons are needed to increase the performance of the network. Moreover, the traditional network's performance dropped significantly when the number of perceptrons was reduced to 7, to match the number of divcon neurons.

Table 3.9 shows the results of option 1 for the letter dataset:

Table 3.9 Option 1 Results (binary sigmoid) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	9.65%	3.9%	4.66%	**10**	16.63%	3.9%	5.58%	**25**	19.45%	3.9%	7.16%
6	11%	3.9%	4.95%	**11**	13.68%	3.9%	5.55%	**30**	18.55%	3.9%	7.46%
7	9.73%	3.9%	5.18%	**12**	10.35%	3.9%	5.47%	**35**	21.4%	3.9%	7.92%
8	9.6%	3.9%	5.29%	**15**	12.75%	3.9%	6.31%	**40**	22.88%	4.35%	8.61%
9	10.93%	3.9%	5.35%	**20**	25.83%	3.9%	6.9%				

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.10 below:

Table 3.10 Option 1 vs. Traditional (binary sigmoid) for Letter Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	61.62%	3.9%	15.94%
Approach 1	6 DNs	11%	3.9%	4.95%
Traditional	6	11.93%	3.9%	4.31%

In this case, as table 3.10 indicates, the performance of the traditional network is significantly higher (**61.62%)** than the network using divcon neurons (**11%**). The case in which the number of perceptrons of the traditional network matches the number of divcon neurons shows a slightly better performance for the traditional network as well.

In the case of *option 2*, table 3.11 illustrates the results for the wine data set:

Table 3.11 Option 2 Results (binary sigmoid) for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	32.43%	88.79%	**14**	100%	32.43%	91.83%	**30**	100%	32.43%	91.83%
6	100%	0%	89.36%	**16**	100%	40.54%	91.93%	**32**	100%	0%	89.36%
7	100%	32.43%	90.99%	**18**	100%	40.54%	91.93%	**34**	100%	0%	87.65%
8	100%	0%	91.42%	**20**	100%	0%	88.29%	**36**	100%	0%	87.15%
9	100%	0%	89.43%	**22**	100%	0%	89.66%	**38**	100%	0%	91.42%
10	100%	40.54%	91.19%	**24**	100%	32.43%	90.36%	**40**	100%	0%	88.96%
11	100%	32.43%	89.62%	**26**	100%	40.54%	88.25%				
12	100%	32.43%	88.62%	**28**	100%	0%	89.86%				

The results of option 2 compared to the traditional network are shown in table 3.12 below:

Table 3.12 Option 2 vs. traditional (binary sigmoid) for wine dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0%	94.35%
Approach 2	5 DNs	100%	32.43%	88.79%
Traditional	5	40.54%	0%	26.53%

As table 3.12 indicates, both the traditional network and Option 2 (divcon neurons throughout the network) are able to provide the same recognition performance of **100%**. In addition, if the number of perceptrons in the traditional network is reduced to match the number of divcon neurons selected, that is, 5 perceptrons in the hidden layer, the performance drops to **40.54%**.

For the vowel data set, table 3.13 shows the results found:

Table 3.13 Option 2 results (binary sigmoid) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	15.8%	9.1%	10.28%	**10**	14.94%	9.1%	10.17%	**24**	18.83%	9.1%	10.93%
6	18.18%	9.1%	11.11%	**11**	14.94%	9.1%	10.53%	**28**	16.67%	9.1%	10.08%
7	18.18%	9.1%	10.4%	**12**	14.94%	9.1%	10.75%	**32**	20.13%	9.1%	10.39%
8	21.21%	9.1%	10.48%	**16**	18.61%	9.1%	10.33%	**36**	19.91%	9.1%	11.22%
9	17.75%	9.1%	10.19%	**20**	18.18%	9.1%	10.77%	**40**	17.32%	9.1%	10.98%

The best performance with the minimum number of DNs in the hidden layer was selected and compared to the traditional network as shown in table 3.14 below:

Table 3.14 Option 2 vs. Traditional (binary sigmoid) for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	68.18%	9.1%	47.74%
Approach 2	8 DNs	21.21%	9.1%	10.48%
Traditional	8	40.54%	0%	26.53%

In this case, the traditional network performs significantly better (**68.18% vs. 21.21%**) than the network that uses divcon neurons in all layers. In addition, if the number of perceptrons in the

traditional network is reduced to match the number of divcon neurons selected, the performance drops to **40.54%**, which still is higher than the performance provided by the divcon neurons.

For the Yeast dataset, table 3.15 shows the results found:

Table 3.15 Option 2 Results (binary sigmoid) for Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	68.86%	39.1%	56.83%	**10**	88.22%	39.1%	55.82%	**24**	88.22%	39.1%	59.03%
6	88.22%	39.1%	56.82%	**11**	88.22%	39.1%	57.78%	**28**	88.22%	39.1%	57.72%
7	88.22%	39.1%	57.83%	**12**	68.86%	39.1%	55.03%	**32**	88.22%	39.1%	57.58%
8	68.86%	39.1%	56.99%	**16**	88.22%	39.1%	55.69%	**36**	68.86%	39.1%	56.57%
9	88.22%	39.1%	57.92%	**20**	88.22%	39.1%	57.75%	**40**	88.22%	39.1%	57.53%

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.16 below:

Table 3.16 Option 2 vs. Traditional (binary sigmoid) for Yeast Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	75.08%	30.05%	69.83%
Approach 2	6 DNs	88.22%	39.1%	56.82%
Traditional	6	39.1%	39.1%	39.1%

For this case, the performance of the traditional network of **75.08%** was surpassed by the use of DNs with **88.22%**. In this case, only 6 divcon neurons were needed to increase the performance of the network. Moreover, when the number of perceptrons in the traditional network was reduced to match the number of DNs, network's performance dropped significantly to **39.1%** correct classifications.

Table 3.17 shows the results of option 2 for the letter dataset:

Table 3.17 Option 2 Results (binary sigmoid) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	54%	3.9%	5.47%	**10**	15.9%	3.9%	4.95%	**25**	15.13%	3.9%	5%
6	30.93%	3.9%	5.22%	**11**	35.38%	3.9%	5.25%	**30**	15.53%	3.9%	5.2%
7	15.25%	3.9%	5.13%	**12**	15.68%	3.9%	5.57%	**35**	15.53%	3.9%	5.18%
8	99.98%	3.9%	6.8%	**15**	61.63%	3.9%	5.53%	**40**	15.03%	3.9%	4.91%
9	30.53%	3.9%	5.54%	**20**	15.13%	3.9%	5.1%				

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.18:

Table 3.18 Option 2 vs. Traditional (binary sigmoid) for Letter Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	61.62%	3.9%	15.94%
Approach 2	8 DNs	**99.98%**	3.9%	6.8%
Traditional	8	17.35%	3.9%	4.74%

As table 3.18 indicates, the performance of the traditional network of **61.62%** was surpassed by using only eight DNs with **99.98%**. Moreover, the number of perceptrons in the traditional network was reduced to match the number of DNs providing a performance of **17.35%** accuracy**.**

3.4.2 Results using a Bipolar Sigmoid Activation Function

The bipolar sigmoid activation function is depicted in Figure 3.4. This function maps its input value *x* within the interval [-1, 1].

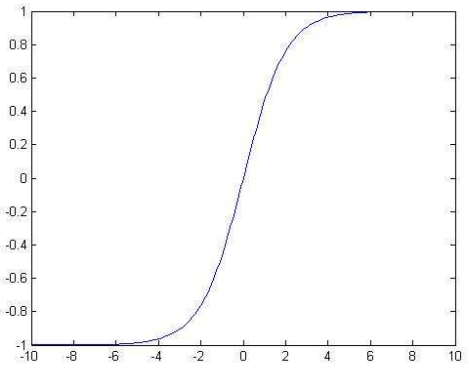


Figure 3.4 Bipolar Sigmoid Activation Function

The mathematical model of the bipolar sigmoid function is shown as follows:

$$\mathbf{y} = (2 / (1 + \exp(-x))) - 1 \qquad (3.1)$$

Table 3.19 shows the results of option 1 for the wine dataset indicating the number of divcon neurons used:

Table 3.19 Option 1 Results (bipolar sigmoid) for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	0%	77.04%	**14**	100%	0%	75.34%	**30**	100%	0%	74..54%
6	100%	0%	74.07%	**16**	100%	0%	75.02%	**32**	100%	0%	73.84%
7	100%	0%	75.31%	**18**	100%	0%	74.61%	**34**	100%	0%	75.58%
8	100%	0%	77.21%	**20**	100%	0%	70.87%	**36**	100%	0%	70.77%
9	100%	0%	74.51%	**22**	100%	0%	77.84%	**38**	100%	0%	70.7%
10	100%	0%	77.51%	**24**	100%	0%	77.08%	**40**	100%	0%	78.14%
11	100%	0%	77.74%	**26**	100%	0%	73.91%				
12	100%	0%	76.64%	**28**	100%	0%	77.01%				

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.20:

Table 3.20 Option 1 vs. Traditional (bipolar sigmoid) for Wine Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0%	61.86%
Approach 1	5 DNs	100%	0%	77.04%
Traditional	5	40.54%	0%	26.53%

As table 3.20 indicates, both the performance of the traditional network and the network using 5 divcon neurons achieve **100%** recognition. In addition, using only 5 perceptrons in the traditional network reduced the performance to **40.54%.**

For the vowel dataset, table 3.21 shows the results found:

Table 3.21 Option 1 Results (bipolar sigmoid) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	63.64%	9.1%	21.36%	**10**	63.64%	9.1%	22.64%	**24**	63.64%	9.1%	24.09%
6	63.64%	9.1%	21.36%	**11**	63.64%	9.1%	24.94%	**28**	100%	9.1%	27.68%
7	54.55%	9.1%	23.86%	**12**	72.73%	9.1%	25.07%	**32**	100%	9.1%	28.68%
8	100%	9.1%	24.08%	**16**	72.73%	9.1%	24.86%	**36**	63.64%	9.1%	25.11%
9	100%	9.1%	27.39%	**20**	54.55%	9.1%	24.02%	**40**	100%	9.1%	25.61%

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.22:

Table 3.22 Option 1 vs. Traditional (bipolar sigmoid) for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	9.1%	27.02%
Approach 1	8 DNs	100%	9.1%	24.08%
Traditional	8	40.54%	0%	26.53%

As table 3.22 indicates, both the performance of the traditional network and the MLP using 8 divcon neurons achieve **100%** recognition. In addition, using only 8 perceptrons in the traditional network used to match the DNs selected provided a decreased performance of **40.54%** accuracy**.**

For the Yeast dataset, table 3.23 shows the results found indicating the number of divcon neurons used in the hidden layer:

Table 3.23 Option 1 Results (bipolar sigmoid) for Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	99.16%	39.1%	55.82%	**10**	99.2%	39.1%	57.23%	**24**	99.2%	39.1%	58.64%
6	88.22%	39.1%	57.44%	**11**	89.06%	39.1%	58.73%	**28**	99.2%	39.1%	59.22%
7	99.2%	39.1%	56.98%	**12**	89.1%	39.1%	56.6%	**32**	89.1%	39.1%	56.44%
8	100%	39.1%	57.1%	**16**	99.2%	39.1%	59.4%	**36**	100%	39.1%	58.57%
9	89.73%	39.1%	57.2%	**20**	88.22%	39.1%	55.75%	**40**	100%	39.1%	57.29%

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.24:

Table 3.24 Option 1 vs. Traditional (bipolar sigmoid) for Yeast Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	99.16%	39.1%	61.08%
Approach 1	8 DNs	100%	39.1%	57.1%
Traditional	8	39.1%	39.1%	39.1%

As table 3.24 indicates, only 8 divcon neurons achieve **100%** recognition, which is higher than the performance of the traditional network. In addition, using only 8 perceptrons in the traditional network used to match the divcon neurons selected provided a decreased performance of **39.1%** recognition**.**

For the Letter dataset, table 3.25 shows the results found indicating the number of divcon neurons used in the hidden layer:

Table 3.25 Option 1 results (bipolar sigmoid) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	61.63%	3.9%	9.84%	**10**	61.63%	3.9%	12.24%	**25**	61.35%	3.9%	12.1%
6	61.33%	3.9%	11.52%	**11**	35.43%	3.9%	9.82%	**30**	61.63%	3.9%	12.57%
7	100%	3.9%	15.13%	**12**	61.63%	3.9%	12.25%	**35**	100%	3.9%	12.12%
8	100%	3.9%	10.43%	**15**	100%	3.9%	13.77%	**40**	100%	3.9%	15.89%
9	100%	3.9%	12.1%	**20**	100%	3.9%	11.51%				

In the case of Option 2, table 3.26 illustrates the results for the Wine dataset, indicating the number of Divcon neurons (DNs) in the hidden layer:

Table 3.26 Option 2 Results (bipolar sigmoid) for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	0%	74.81%	**14**	100%	0%	76.91%	**30**	100%	0%	74.54%
6	100%	0%	84.38%	**16**	100%	0%	78.34%	**32**	100%	32.43%	87%
7	100%	0%	76.11%	**18**	100%	0%	91.02%	**34**	100%	0%	75.58%
8	100%	32.48%	84.48%	**20**	100%	32.43%	85.02%	**36**	100%	0%	83.22%
9	100%	0%	84.15%	**22**	100%	0%	77.84%	**38**	100%	0%	70.7%
10	100%	0%	78.61%	**24**	100%	0%	83.28%	**40**	100%	0%	82.32%
11	100%	0%	84.65%	**26**	100%	0%	73.91%				
12	100%	0%	83.4%	**28**	100%	0%	80.11%				

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.27:

Table 3.27 Option 2 vs. Traditional (bipolar sigmoid) for Wine Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0%	61.86%
Approach 2	5 DNs	100%	0%	74.81%
Traditional	5	40.54%	0%	26.53%

As table 3.27 indicates, both the traditional network and the divcon neurons achieve **100%** recognition. However, only 5 divcon neurons are required to achieve the same performance as the traditional network. In addition, using only 5 perceptrons in the traditional network to match the divcon neurons selected provided a decreased performance of **40.54%** recognition**.**

For the Vowel dataset, table 3.28 shows the results found indicating the number of Divcon neurons used in the hidden layer:

Table 3.28 Option 2 Results (bipolar sigmoid) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	15.8%	9.1%	10.28%	**10**	14.94%	9.1%	10.17%	**24**	18.83%	9.1%	10.93%
6	18.18%	9.1%	11.11%	**11**	14.94%	9.1%	10.53%	**28**	16.67%	9.1%	10.08%
7	18.18%	9.1%	10.4%	**12**	14.94%	9.1%	10.75%	**32**	20.13%	9.1%	10.39%
8	21.21%	9.1%	10.48%	**16**	18.61%	9.1%	10.33%	**36**	19.91%	9.1%	11.22%
9	17.75%	9.1%	10.19%	**20**	18.18%	9.1%	10.77%	**40**	17.32%	9.1%	10.98%

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.29:

Table 3.29 Option 2 vs. Traditional (bipolar sigmoid) for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	9.1%	27.02%
Approach 2	11 DNs	100%	9.1%	13.48%
Traditional	11	100%	9.1%	33.17%

In this particular case, both the traditional network and the divcon neurons provide a performance of **100%** recognition. Moreover, the number of perceptrons in the hidden layer of the traditional network was reduced to match the number of divcon neurons selected and the same performance was achieved. Therefore, in this particular case the performance of the traditional network did not decrease.

For the Yeast dataset, table 3.30 shows the results found indicating the number of divcon neurons used in the hidden layer:

Table 3.30 Option 2 Results (bipolar sigmoid) for Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	88.55%	39.1%	58.8%	**10**	88.2%	39.1%	60.59%	**24**	88.72%	39.1%	63.79%
6	88.22%	39.1%	59.4%	**11**	88.22%	39.1%	60.78%	**28**	88.22%	39.1%	53.65%
7	88.22%	39.1%	60.57%	**12**	88.22%	39.1%	60.78%	**32**	88.77%	39.1%	62.3%
8	88.22%	39.1%	61.29%	**16**	88.22%	39.1%	60.15%	**36**	88.22%	39.1%	59.13%
9	88.22%	39.1%	59.55%	**20**	88.22%	39.1%	61.1%	**40**	88.22%	39.1%	62.14%

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.31:

Table 3.31 Option 2 vs. Traditional (bipolar sigmoid) for Yeast Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	99.16%	39.1%	61.08%
Approach 2	5 DNs	88.55%	39.1%	58.8%
Traditional	5	99.16%	39.1%	65%

As table 3.31 indicates, the traditional network provides a performance of **99.16%** recognition, which is higher than the performance provided by the divcon neurons (**88.55%**). In addition, only 5 perceptrons in the hidden layer of the traditional network provided a higher performance than the same number of divcon neurons.

For the Letter dataset, table 3.32 shows the results found indicating the number of divcon neurons used in the hidden layer:

Table 3.32 Option 2 Results (bipolar sigmoid) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	31.25%	3.9%	10.19%	**10**	53.98%	3.9%	10.04%	**25**	31.25%	3.9%	7.2%
6	30.83%	3.9%	10.55%	**11**	61.35%	3.9%	8.98%	**30**	61.85%	3.9%	8.41%
7	61.63%	3.9%	9.99%	**12**	61.35%	3.9%	8.98%	**35**	37.75%	3.9%	7.16%
8	62%	3.9%	9.91%	**15**	42.1%	3.9%	8.74%	**40**	30.53%	3.9%	7.99%
9	18.65%	3.9%	7.56%	**20**	49.7%	3.9%	8.19%				

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.33:

Table 3.33 Option 2 vs. Traditional (bipolar sigmoid) for Letter Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	61.63%	3.9%	20.1%
Approach 2	8 DNs	62%	3.9%	9.91%
Traditional	8	17.35%	3.9%	4.63%

As table 3.33 indicates, the traditional network and the DNs provide a similar performance of **61.63%** and **62%** respectively. In addition, by using only 8 perceptrons in the traditional network to match the number of DNs selected, the performance decreased significantly to **17.35%**.

3.4.3 Results using a Gaussian Activation Function

The Gaussian activation function is depicted in Figure 3.5:

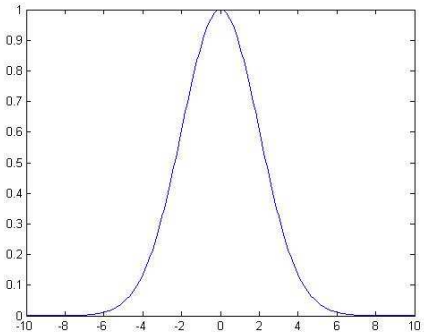


Figure 3.5 Gaussian Activation Function

This function maps its input value *x* within the interval [-1, 1]. Mathematically, the Gaussian function is described as follows:

$$\mathbf{y} = \exp^{((-x\wedge 2)/2)} \tag{3.2}$$

Table 3.34 shows the results of approach 1 for the Wine dataset:

Table 3.34 Option 1 Results (Gaussian) for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	0%	35.27%	**14**	32.43%	0%	23.62%	**30**	32.43%	0%	16.42%
6	100%	0%	29.43%	**16**	100%	0%	18.05%	**32**	32.43%	0%	16.42%
7	100%	0%	16.15%	**18**	32.43%	0%	19.22%	**34**	32.43%	0%	19.22%
8	100%	0%	24.22%	**20**	32.43%	0%	18.42%	**36**	32.43%	0%	16.01%
9	100%	0%	18.55%	**22**	32.43%	0%	19.25%	**38**	32.43%	0%	12.01%
10	100%	0%	13.68%	**24**	32.43%	0%	18.42%	**40**	32.43%	0%	12.01%
11	32.43%	0%	20.42%	**26**	32.43%	0%	18.42%				
12	32.43%	0%	16.01%	**28**	100%	0%	18.85%				

The best performance with the minimum number of divcon neurons in the hidden layer is selected and compared to the traditional network as shown in table 3.35:

Table 3.35 Option 1 vs. Traditional (Gaussian) for Wine Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0%	58.1%
Approach 1	5 DNs	100%	0%	35.27%
Traditional	5	100%	0%	57.26%

As table 3.35 indicates, both the traditional network and the DNs provide a performance of **100%** recognition. However, by matching the number of perceptrons in the traditional network to the number of DNs, the same performance of 100% recognition was achieved.

Table 3.36 shows the results of option 1 for the Vowel dataset:

Table 3.36 Option 1 Results (Gaussian) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	9.1%	9.1%	9.1%	**10**	9.1%	9.1%	9.1%	**24**	9.1%	9.1%	9.1%
6	9.1%	9.1%	9.1%	**11**	9.1%	9.1%	9.1%	**28**	9.1%	9.1%	9.1%
7	9.1%	9.1%	9.1%	**12**	9.1%	9.1%	9.1%	**32**	9.1%	9.1%	9.1%
8	9.1%	9.1%	9.1%	**16**	9.1%	9.1%	9.1%	**36**	9.1%	9.1%	9.1%
9	9.1%	9.1%	9.1%	**20**	9.1%	9.1%	9.1%	**40**	9.1%	9.1%	9.1%

The best performance with the minimum number of Divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.37:

Table 3.37 Option 1 vs. Traditional (Gaussian) for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	9.1%	9.1%	9.1%
Approach1	5 DNs	9.1%	9.1%	9.1%
Traditional	5	9.1%	9.1%	9.1%

As table 3.37 indicates, the performance of the traditional network and the DNs is the same (9.1%).

Table 3.38 shows the results of option 1 for the Yeast dataset:

Table 3.38 Option 1 Results (Gaussian) for Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	68.7%	39.1%	61.03%	**10**	68.7%	39.1%	60.67%	**24**	68.7%	39.1%	58.83%
6	68.7%	39.1%	60.66%	**11**	68.7%	39.1%	61.39%	**28**	68.7%	39.1%	55.55%
7	68.7%	39.1%	60.66%	**12**	68.7%	39.1%	60.33%	**32**	68.7%	39.1%	59.93%
8	68.7%	39.1%	60.66%	**16**	68.7%	39.1%	60.66%	**36**	68.7%	39.1%	58.15%
9	68.7%	39.1%	61.39%	**20**	68.7%	39.1%	61.39%	**40**	68.7%	39.1%	58.11%

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.39:

Table 3.39 Option 1 vs. Traditional (Gaussian) for Yeast Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	68.7%	39.1%	45.95%
Approach1	5 DNs	68.7%	39.1%	61.03%
Traditional	5	68.7%	39.1%	59.9%

As table 3.39 indicates, the performance of the traditional network and the DNs is the same.

Table 3.40 shows the results of option 1 for the Letter dataset:

Table 3.40 Option 1 Results (Gaussian) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	7.3%	3.9%	4.28%	**10**	7.3%	3.9%	4.28%	**25**	7.3%	3.9%	4.24%
6	7.3%	3.9%	4.32%	**11**	7.3%	3.9%	4.28%	**30**	7.3%	3.9%	4.32%
7	7.3%	3.9%	4.32%	**12**	7.3%	3.9%	4.24%	**35**	7.3%	3.9%	4.28%
8	7.3%	3.9%	4.32%	**15**	7.3%	3.9%	4.28%	**40**	7.3%	3.9%	4.28%
9	7.3%	3.9%	4.28%	**20**	7.3%	3.9%	4.32%				

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.41:

Table 3.41 Option 1 vs. Traditional (Gaussian) for Letter Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	7.3%	3.9%	4.32%
Approach1	5 DNs	7.3%	3.9%	4.28%
Traditional	5	7.3%	3.9%	4.32%

As table 3.41 indicates, in all cases the traditional network and the divcon neurons provided the same performance (7.3%).

In the case of option 2, the results for the Wine dataset are shown in table 3.42:

Table 3.42 Option 2 Results (Gaussian) for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	0%	33.67%	**14**	32.43%	0%	21.22%	**30**	32.43%	0%	32.02%
6	100%	0%	30.86%	**16**	100%	0%	35.33%	**32**	100%	0%	29.36%
7	100%	0%	27.79%	**18**	100%	0%	24.06%	**34**	100%	0%	29.46%
8	100%	0%	37.34%	**20**	100%	0%	24.96%	**36**	100%	0%	29.13%
9	100%	0%	28.23%	**22**	100%	0%	23.29%	**38**	32.43%	0%	20.82%
10	100%	0%	29.53%	**24**	100%	0%	30.83%	**40**	100%	0%	26.59%
11	100%	0%	32%	**26**	100%	0%	22.55%				
12	100%	0%	25.36%	**28**	100%	0%	20.52%				

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.43:

Table 3.43 Option 2 vs. Traditional (Gaussian) for Wine Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0.0%	58.1%
Approach 2	5 DNs	100%	0.0%	33.67%
Traditional	5	100%	0.0%	57.26%

As table 3.43 indicates, both the traditional network and 5 divcon neurons provide a performance of **100%** recognition. In addition, when the number of perceptrons in the traditional network is reduced to match the number of divcon neurons selected, the performance of the network provided **100%** recognition as well.

Table 3.44 shows the results for the Vowel dataset using Option 2:

Table 3.44 Option 2 Results (Gaussian) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	18.18%	9.1%	9.21%	**10**	9.1%	9.1%	9.1%	**24**	18.18%	9.1%	9.26%
6	9.1%	9.1%	9.1%	**11**	9.1%	9.1%	9.1%	**28**	9.3%	9.1%	9.11%
7	9.1%	9.1%	9.1%	**12**	9.1%	9.1%	9.1%	**32**	18.18%	9.1%	9.22%
8	9.1%	9.1%	9.1%	**16**	9.52%	9.1%	9.11%	**36**	9.1%	9.1%	9.1%
9	9.1%	9.1%	9.1%	**20**	18.18%	9.1%	9.22%	**40**	9.1%	9.1%	9.11%

The best performance with the minimum number of divcon neurons in the hidden layer was selected and compared to the traditional network as shown in table 3.45:

Table 3.45 Option 2 vs. Traditional (Gaussian) for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	9.1%	9.1%	9.1%
Approach2	5 DNs	18.18%	9.1%	9.21%
Traditional	5	9.1%	9.1%	9.1%

As table 3.45 indicates, the performance of the traditional network (9.1%) was lower than the divcon neurons' (18.18%). In addition, by matching the number of perceptrons in the traditional network to the number of divcon neurons selected, the performance of the traditional network provided was 9.1% recognition as well.

Table 3.46 shows the corresponding results for the Yeast dataset:

Table 3.46 Option 2 Results (Gaussian) for Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	68.69%	39.1%	59.92%	**10**	68.69%	39.1%	58.1%	**24**	68.69%	39.1%	52.62%
6	68.69%	39.1%	60.65%	**11**	88.22%	39.1%	59.8%	**28**	68.69%	39.1%	51.89%
7	68.69%	39.1%	58.83%	**12**	68.69%	39.1%	55.9%	**32**	68.69%	39.1%	50.79%
8	68.69%	39.1%	57.37%	**16**	68.69%	39.1%	55.9%	**36**	68.69%	39.1%	51.16%
9	68.69%	39.1%	59.56%	**20**	68.69%	39.1%	53.71%	**40**	68.69%	39.1%	51.16%

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.47:

Table 3.47 Option 2 vs. Traditional (Gaussian) for Yeast Dataset

MLP	#DNs	Max	Min	Average
Traditional	22	68.7%	39.1%	45.95%
Approach2	11 DNs	88.22%	39.1%	59.8%
Traditional	11	68.7%	39.1%	55.89%

As table 3.47 indicates, the performance of the divcon neurons (88.22%) was higher than the traditional network (68.7%). In addition, by matching the number of perceptrons in the traditional

network to the number of divcon neurons selected, the performance of the traditional network provided was 68.7% recognition as well.

Table 3.48 shows the corresponding results for the Letter dataset:

Table 3.48 Option 2 Results (Gaussian) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	7.3%	3.9%	4.15%	**10**	30.43%	3.9%	5.88%	**25**	15%	3.9%	4.2%
6	7.3%	3.9%	4.33%	**11**	7.3%	3.9%	4.15%	**30**	7.3%	3.9%	4.19%
7	7.3%	3.9%	4.24%	**12**	7.3%	3.9%	4.4%	**35**	7.3%	3.9%	4.28%
8	7.3%	3.9%	4.15%	**15**	7.3%	3.9%	4.1%	**40**	30.5%	3.9%	5.91%
9	15.53%	3.9%	4.55%	**20**	30.43%	3.9%	4.61%				

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.49:

Table 3.49 Option 2 vs. Traditional (Gaussian) for Letter Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	7.3%	3.9%	4.32%
Approach2	10 DNs	30.43%	3.9%	5.88%
Traditional	10	7.3%	3.9%	4.32%

As table 3.49 indicates, the performance of the divcon neurons (30.43%) is higher than the traditional network (7.3%). In addition, by matching the number of perceptrons in the traditional network to the number of divcon neurons selected, the performance of the traditional network provided was 7.3% recognition as well. Consequently, the divcon neurons were able to increase the performance of the traditional network.

3.4.4 Results using a Wavelet Activation Function

The Wavelet activation function is depicted in Figure 3.6:

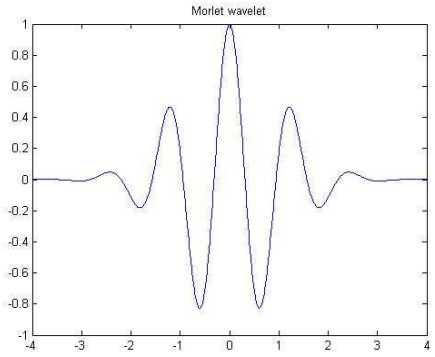


Figure 3.6 Wavelet Activation Function

This function maps its input value *x* within the interval [-1, -1]. Mathematically, the wavelet function is described as follows:

$$\mathbf{y} = \cos(1.75x_s) * \exp(-x_s^2/2) \qquad (3.3)$$

Table 3.50 shows the results of option 1 for the wine dataset using a wavelet function:

Table 3.50 Option 1 Results (Wavelet) for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	0%	7.96%	**14**	39.89%	0%	2.87%	**30**	100%	0%	6.57%
6	100%	0%	7.63%	**16**	39.89%	0%	5.58%	**32**	100%	0%	4.02%
7	39.89%	0%	5%	**18**	39.89%	0%	2.38%	**34**	39.89%	0%	3.74%
8	100%	0%	7.71%	**20**	39.89%	0%	1.89%	**36**	39.89%	0%	2.46%
9	39.89%	0%	5.65%	**22**	100%	0%	23.29%	**38**	39.89%	0%	2.46%
10	39.89%	0%	5.49%	**24**	39.89%	0%	4.43%	**40**	39.89%	0%	0.49%
11	39.89%	0%	7.05%	**26**	39.89%	0%	1.31%				
12	39.89%	0%	5.83%	**28**	39.89%	0%	1.39%				

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.51:

Table 3.51 Option 1 vs. Traditional (Wavelet) for Wine Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0.0%	9.94%
Approach2	5 DNs	100%	0.0%	7.96%
Traditional	5	32.43%	0.0%	10.81%

As table 3.51 indicates, the performance of the divcon neurons and the traditional network is the same (100%). However, only 5 divcon neurons are needed to achieve the same performance. In addition, by matching the number of perceptrons in the traditional network to the number of divcon neurons selected, the performance of the traditional network dropped to 32.43% recognition.

Regarding the Vowel dataset, table 3.52 shows the results of approach 1 using a wavelet function:

Table 3.52 Option 1 Results (Wavelet) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	18.18%	9.1%	9.24%	**10**	16.67%	9.1%	9.24%	**24**	9.5%	9.1%	9.2%
6	13.64%	9.1%	9.18%	**11**	14.4%	9.1%	9.18%	**28**	18%	9.1%	9.25%
7	10.61%	9.1%	9.13%	**12**	17.42%	9.1%	9.28%	**32**	18.18%	9.1%	9.39%
8	18.18%	9.1%	9.21%	**16**	12.5%	9.1%	9.18%	**36**	18.18%	9.1%	9.41%
9	12.7%	9.1%	9.17%	**20**	13.83%	9.1%	9.23%	**40**	18.94%	9.1%	9.46%

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.53:

Table 3.53 Option 1 vs. Traditional (Wavelet) for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	37.23%	9.1%	13.04%
Approach2	5 DNs	18.18%	9.1%	9.24%
Traditional	5	32.43%	0.0%	10.81%

As table 3.53 indicates, the traditional network performs better than the divcon neurons. In addition, by matching the number of perceptrons in the traditional network to the number of divcon neurons selected, the performance of the traditional network is still higher than the divcon neurons.

With regard to the Yeast dataset, table 3.54 shows the results of option 1 using a wavelet function:

Table 3.54 Option 1 Results (Wavelet) for Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	54.4%	25.96%	27.39%	**10**	54.4%	25.96%	27.11%	**24**	54.4%	25.96%	26.69%
6	54.4%	25.96%	27.63%	**11**	54.4%	25.96%	27.04%	**28**	54.4%	25.96%	26.8%
7	54.4%	25.96%	27.31%	**12**	54.4%	25.96%	27.01%	**32**	54.4%	25.96%	26.3%
8	54.4%	25.96%	26.94%	**16**	54.4%	25.96%	27.01%	**36**	54.4%	25.96%	26.31%
9	54.4%	25.96%	27.02%	**20**	54.4%	25.96%	27.41%	**40**	54.4%	25.96%	28.38%

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.55:

Table 3.55 Option 1 vs. Traditional (Wavelet) for Yeast Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	39.1%	39.1%	39.1%
Approach2	5 DNs	54.4%	25.96%	27.39%
Traditional	5	46.63%	39.1%	39.48%

As table 3.55 indicates, the performance of the divcon neurons (54.4%) is higher than the network (39.1%). In addition, by matching the number of perceptrons in the traditional network to the number of divcon neurons selected, the performance of the traditional network is still lower than the divcon neurons.

With regard to the Letter dataset, table 3.56 shows the results of option 1 using a wavelet function:

Table 3.56 Option 1 Results (Wavelet) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	7.67%	3.96%	4.01%	**10**	3.96%	3.96%	3.96%	**25**	7.81%	3.96%	4.01%
6	7.67%	3.96%	4.01%	**11**	3.96%	3.96%	3.96%	**30**	7.81%	3.96%	4.1%
7	3.96%	3.96%	3.96%	**12**	7.64%	3.96%	4.1%	**35**	7.81%	3.96%	4.29%
8	7.81%	3.96%	4.01%	**15**	7.8%	3.96%	4.45%	**40**	7.79%	3.96%	4.01%
9	3.96%	3.96%	3.96%	**20**	7.69%	3.96%	4.01%				

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.57:

Table 3.57 Option 1 vs. Traditional (Wavelet) for Letter Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	61.35%	3.9%	7.98%
Approach2	8 DNs	7.81%	3.96%	4.01%
Traditional	8	61.35%	3.9%	9.72%

As table 3.57 indicates, the performance of the divcon neurons (7.81%) is lower than the network (61.35%). In addition, by matching the number of perceptrons in the traditional network to the number of divcon neurons selected, the performance of the traditional network is still higher than the divcon neurons.

For option 2, table 3.58 shows the results for the Wine dataset using a wavelet function:

Table 3.58 Option 2 Results (Wavelet) for Wine Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	100%	0%	23.96%	**14**	100%	0%	10.94%	**30**	100%	0%	9.84%
6	100%	0%	17.98%	**16**	100%	0%	6.47%	**32**	100%	0%	20.45%
7	100%	0%	12.85%	**18**	100%	0%	13.65%	**34**	40.54%	0%	4.3%
8	100%	0%	11.24%	**20**	100%	0%	22.49%	**36**	40.54%	0%	7.1%
9	100%	0%	10.91%	**22**	100%	0%	13.05%	**38**	100%	0%	10.24%
10	100%	0%	16.55%	**24**	100%	0%	12.01%	**40**	100%	0%	17.98%
11	100%	0%	18.18%	**26**	100%	0%	8.98%				
12	100%	0%	10.94%	**28**	100%	0%	12.15%				

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.59:

Table 3.59 Option 2 vs. Traditional (Wavelet) for Wine Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	100%	0.0%	9.94%
Approach2	5 DNs	100%	0.0%	23.96%
Traditional	5	32.43%	0.0%	10.81%

As table 3.59 indicates, both the divcon neurons and the traditional network provide a performance of 100% recognition. In addition, by matching the number of perceptrons in the traditional

network to the number of divcon neurons selected, the performance of the traditional network dropped to 32.43% recognition.

Table 3.60 shows the results for the Vowel dataset using a wavelet function using option 2:

Table 3.60 Option 2 Results (Wavelet) for Vowel Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	54.55%	9.1%	12.08%	**10**	63.64%	9.1%	12.3%	**24**	18.18%	9.1%	9.74%
6	36.36%	9.1%	12.13%	**11**	36.36%	9.1%	10.53%	**28**	35.5%	9.1%	11.23%
7	28.36%	9.1%	10.64%	**12**	36.36%	9.1%	12.1%	**32**	36.36%	9.1%	10.32%
8	22.1%	9.1%	10.44%	**16**	57.8%	9.1%	11.99%	**36**	36.36%	9.1%	11.62%
9	36.36%	9.1%	10.79%	**20**	36.36%	9.1%	10.72%	**40**	72.73%	9.1%	12.82%

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.61:

Table 3.61 Option 2 vs. Traditional (Wavelet) for Vowel Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	37.23%	9.1%	13.04%
Approach2	5 DNs	54.55%	9.1%	12.08%
Traditional	5	36.36%	9.1%	13.55%

As table 3.61 indicates, the DNs provide a higher performance (54.55%) when compared to the traditional network (37.23%). In addition, by matching the number of perceptrons in the traditional network to the number of DNs, the performance dropped to 36.36% recognition.

Table 3.62 shows the results for the Yeast dataset using a wavelet function using option 2:

Table 3.62 Option 2 Results (Wavelet) for Yeast Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	68.69%	39.1%	46.23%	**10**	75.42%	39.1%	42.76%	**24**	68.69%	39.1%	40.62%
6	68.69%	39.1%	42.84%	**11**	68.69%	39.1%	41.97%	**28**	88.22%	39.1%	40.81%
7	88.22%	39.1%	41.98%	**12**	68.69%	39.1%	41.28%	**32**	64.14%	39.1%	40.66%
8	69.53%	39.1%	41.52%	**16**	68.69%	39.1%	43.12%	**36**	55.9%	39.1%	40.31%
9	62.46%	39.1%	41.87%	**20**	68.69%	39.1%	41.3%	**40**	68.69%	39.1%	40.97%

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.63:

Table 3.63 Option 2 vs. Traditional (Wavelet) for Yeast Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	39.1%	39.1%	39.1%
Approach2	7 DNs	88.22%	39.1%	41.98%
Traditional	5	46.63%	39.1%	39.48%

As table 3.63 indicates, the DNs provide a higher performance (88.22%) when compared to the traditional network (39.1%). In addition, by matching the number of perceptrons in the traditional network to the number of DNs, the performance resulted in 46.63% recognition.

Table 3.64 shows the results for the Yeast dataset using a wavelet function using option 2:

Table 3.64 Option 2 Results (Wavelet) for Letter Dataset

#DNS	Max	Min	Avg	#DNS	Max	Min	Avg	#DNS	Max	Min	Avg
5	15.5%	3.9%	5.41%	**10**	15.83%	3.9%	4.92%	**25**	15%	3.9%	4.19%
6	15.43%	3.9%	4.75%	**11**	15.53%	3.9%	4.23%	**30**	31.1%	3.9%	4.71%
7	8.1%	3.9%	4.14%	**12**	15.83%	3.9%	4.44%	**35**	8%	3.9%	4.09%
8	15.43%	3.9%	4.47%	**15**	15%	3.9%	4.33%	**40**	15.23%	3.9%	4.18%
9	15.53%	3.9%	4.64%	**20**	8%	3.9%	4.14%				

The best performance with the minimum number of divcon neurons was selected and compared to the traditional network as shown in table 3.65:

Table 3.65 Option 2 vs. Traditional (Wavelet) for Letter Dataset

MLP	# DNs	Max	Min	Average
Traditional	22	61.35%	3.9%	7.98%
Approach2	8 DNs	15.43%	3.9%	4.47%
Traditional	8	61.35%	3.9%	9.72%

As table 3.65 indicates, the traditional network performs significantly better than the DNs, even if the number of perceptrons in the traditional network matches the number of DNs.

Based on the results obtained, the following conclusions were drawn and are listed below:

- The best performance was obtained using the bipolar sigmoid function
- In general, Option 1 performs better than Option 2

- Regarding the traditional network's performance, there is not a lot of room for improvement

Consequently, the bipolar sigmoid function is selected as the activation function to be used in the next phase of the research. In addition, only Option 1 is selected to continue with phase two of the development of the DN model considering that Option 2 does not provide additional benefits. Considering these conclusions, Table 3.66 summarizes the performance of the traditional network using the bipolar sigmoid function and Table 3.67 shows a summary of the results found using the initial DN model with bipolar sigmoid function from both Option 1 and Option 2:

Table 3.66 Traditional Network Results with Bipolar Sigmoid Function

Dataset	Max	Min	Avg
Wine	100%	0.0%	61.86%
Vowel	100%	9.1%	27.02%
Yeast	99.16%	39.1%	61.08%
Letter	61.63%	3.9%	20.1%

Table 3.67 Initial DN Model (Bipolar Sigmoid) Results (both approaches)

Dataset	#DNs	Max-1	Min-1	Avg-1	#DNs	Max-2	Min-2	Avg-2
Wine	5	100%	0.0%	77.04%	5	100%	0.0%	61.86%
Vowel	8	100%	9.1%	24.08%	11	100%	9.1%	13.48%
Yeast	8	100%	39.1%	57.1%	5	88.55%	39.1%	58.8%
Letter	7	100%	3.9%	15.13%	8	62.0%	3.9%	9.91%

3.5 Divcon Neuron Model Development-Phase Two

In the attempt to improve the initial DN model, several models were developed, implemented, tested, and compared to the traditional network, which are described in detail in the following sections.

3.5.1 XOR Units Model

A model based on the oversimplified behaviour of a biological neuron was developed using the following rules:

1) *If product of w * x is +, and w > 1, then product = 1*

2) *If product of w * x is +, and w < 1, then product = -1*

3) *If product of w * x is +, and w = 1, then product = 1*

4) *If product of w * x is -, and |w| > 1, then product = -1*

5) *If product of w * x is -, and |w| < 1, then product = 1*

6) *If product of w * x is -, and |w| = 1, then product = -1*

These rules monitor the sign of the product of each input x, its corresponding weight w, and the magnitude of the weights. Based on this type of evaluation, the product is determined to be excitatory (value of 1) or inhibitory (value of -1). This behaviour can be translated into a table, which ends up being a simple XOR operation (ignoring the case when $w = 1$), expressed in bipolar notation, as shown in Figure 3.7, where p is the sign of the product of the input times its corresponding weight and $|w|$ is the magnitude of the weight.

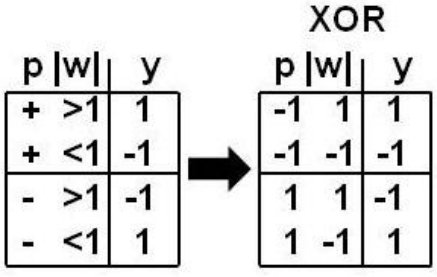


Figure 3.7 XOR Units Model Behaviour

Based on this behavioral description, the mathematical model of the DN for an input vector pattern of length p is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (x_{ik} \wedge |w_{jk}|) \tag{3.4}$$

$$z_{j2} = \sum_{k=\frac{p}{2}}^{p-1} (x_{ik} \wedge |x_{jk}|) + \text{b} \tag{3.5}$$

$$Y_j = f((z_{j1} * v_{j1}) + (z_{j2} * v_{j2}) + v_{j3}) \tag{3.6}$$

Where x_{ik} represents the sign of the product of the k^{th} input of vector pattern i, and its corresponding weight w_{jk}, $|w_{jk}|$ is the magnitude of the weight corresponding to the link between input k and neuron j, b is a bias term, z_{j1} is the summation of the XOR operations between x_{ik} and $|w_{jk}|$ for the first half of the inputs, z_{j2} is the summation of the XOR operations of the remaining half of the inputs, v_{j1} and v_{j2} are internal weights of neuron j, v_{j3} is a bias term of neuron j, $f(x)$ is an activation function, and Y_j is the output of neuron j. Figure 3.8 shows the proposed XOR units model:

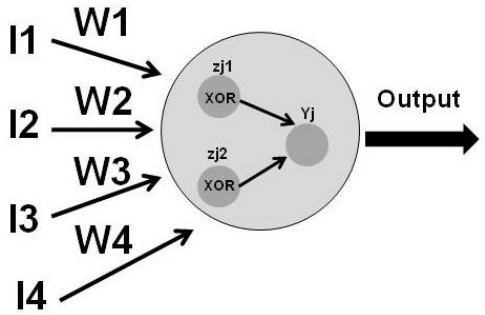


Figure 3.8 DN model with XOR Units

Comparing this model to the initial design, it can readily be observed that the XOR Units model is computationally lighter than the initial DN model. Specifically, two activation functions are eliminated, and the multiplication of the inputs is substituted by a simple XOR operation.

3.5.2 XOR Units Model Results

Using the bipolar sigmoid function, Table 3.68 shows the results of the network using DNs composed of XOR units indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.68 Results of MLP with DNs Composed of XOR Units

Benchmark	# DNs	Max	Min	Avg
Wine	5	100%	0.0%	81.28%
Vowel	7	54.11%	9.1%	15.53%
Yeast	5	100%	39.1%	61.13%
Letter	7	100%	3.9%	12.1%

In this case, the performance related to the Wine, Yeast, and Letter benchmark problems remains the same as the initial DN design. However, there is an improvement in the number of DNs used for the yeast data set (only five). The exception is the performance evaluating the Vowel dataset, which is significantly lower than both the initial design and the traditional network. In general, this model performs better than the traditional network.

3.5.3 Shift Units Model

Another model of the DN was developed using the following rules:

1) *If product of w * x is +, and w > 1, then do a left shift*

2) *If product of w * x is +, and w < 1, then do a right shift*

3) *If product of w * x is +, and w = 1, then no change to input*

4) *If product of w * x is -, and |w| > 1, then do a left shift*

5) *If product of w * x is -, and |w| < 1, then do a right shift*

6) *If product of w * x is -, and |w| = 1, then no change to input*

These rules monitor the sign of the product of each input *x*, its corresponding weight *w*, and the magnitude of the weights |*w*|. Considering that the operation is performed on the input, the shifts basically either increase (left shift) or decrease (right shift) the value, based on the associated weight. This behaviour can be translated into a table as shown in Figure 3.9 below:

p	|w|	y
+	>1	LS
+	<1	RS
+	=1	NOP
-	>1	LS
-	<1	RS
-	=1	NOP

Figure 3.9 LS/RS Behavior

For this case, a left shift (LF)/right shift (RS) operation or no operation (NOP) is used to eliminate the actual multiplication of the input and its corresponding weight. Based on this behavior, the mathematical model of the DN for an input vector pattern of length *p* is shown below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (\text{SH/NOP } (x_{ik,j}) \tag{3.7}$$

$$z_{j2} = \sum_{k=\frac{p}{2}}^{p-1} (\text{SH/NOP } (x_{ik,j}) + \text{ b} \tag{3.8}$$

$$Y_j = f((z_{j1} * v_{j1}) + (z_{j2} * v_{j2}) + v_{j3}) \tag{3.9}$$

Where $x_{ik,j}$ represents the shifted (SH) or intact (NOP) input *k* of vector pattern *i*, connected to neuron *j* based on the rules previously described. z_{j1} is the summation of the first half of the shifted/intact inputs, z_{j2} is the summation of the remaining half of the shifted/intact inputs, *b* is a

bias term, v_{j1} and v_{j2} are the internal weights of neuron j, v_{j3} is a bias term of neuron j, $f(x)$ is an activation function, and Y_j is the output of neuron j.

The structure of this model shown in Figure 3.10 is similar to that of the XOR-units model:

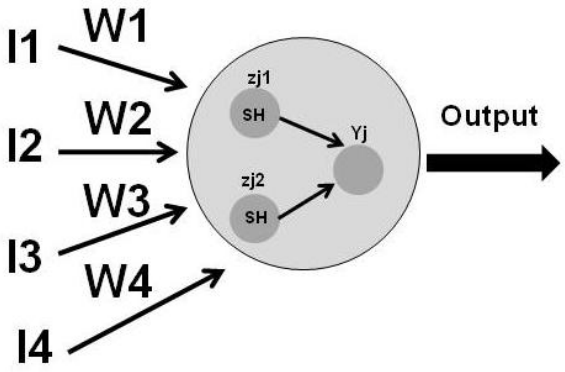


Figure 3.10 DN model with Shift Units

Comparing this model to the initial design, two activation functions are eliminated, and the multiplication of the inputs is substituted by a simple shift operation.

3.5.4 Shift Units Model Results

Using the bipolar sigmoid function, Table 3.69 shows the results of the network using DNs composed of *Shift* units indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.69 Results of MLP with DNs Composed of Shift Units

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	0.0%	74.84%
Vowel	11	98.92%	9.1%	22.56%
Yeast	5	99.16%	39.1%	63.41%
Letter	10	100%	3.9%	14.6%

In this case, the performance related to the Wine and Letter datasets remains the same as the initial DN. However, the Letter dataset requires more neurons. With regard to the Vowel and Yeast datasets, there is about 1% difference in the performance compared to the initial DN design. However, there is an improvement in the number of DNs used for the Yeast dataset (only five) and more neurons were required for the Vowel dataset (eleven). In general, this model performs better than the traditional network.

3.5.5 Quantum Neurons Model

An additional hybrid design of the DN was developed using quantum neurons [CAO09, LIF09, MOR06, XIA09, and ZHA06]. In this case, the weights are converted to *qubits* and are described as a two-component matrix represented as [a b]. Given an input vector pattern of length *p*, the mathematical model of the hybrid DN is described as follows [XIA09]:

$$z_{j1} = [1\ 1]^T . \sum_{k=0}^{\left(\frac{p}{2}\right)-1} x_{ik} | \emptyset_{\mathrm{jk}} > \tag{3.10}$$

$$z_{j2} = [1\ 1]^T . \sum_{k=p/2}^{p} x_{ik} | \emptyset_{\mathrm{jk}} > \tag{3.11}$$

$$Y_j = f[(z_{j1} * v_{j1}) + (z_{j2} * v_{j2}) + v_{j3}] \tag{3.12}$$

Where x_{ik} is the input *k* of vector pattern *i*, $\emptyset_{jk}$ is the weight coefficient *k* (converted to qbit) of the corresponding input *k* connected to DN *j*, z_{j1} is evaluation of the first half of the inputs, z_{j2} is the evaluation of the remaining half of the inputs, v_{j1} and v_{j2} are internal weights of neuron *j*, v_{j3} is a bias term of neuron *j*, *f(x)* is an activation function, and Y_j is the output of neuron *j*. The structure of this model is depicted in Figure 3.11:

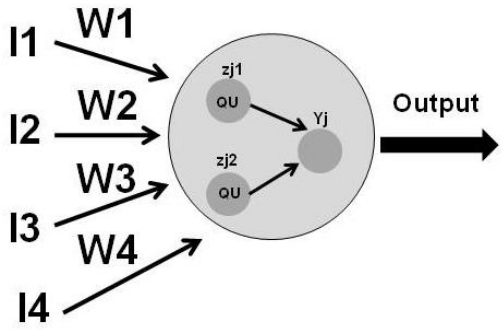


Figure 3.11 DN model with Quantum Neurons

Comparing this model to the initial design, two activation functions are eliminated and the size of the weight matrix doubled.

3.5.6 Quantum Neurons Model Results

Using the bipolar sigmoid function, Table 3.70 shows the results of the network using DNs composed of quantum neurons indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.70 Results of MLP with DNs Composed of Quantum Neurons

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	32.43%	92.99%
Vowel	8	100%	9.1%	17.33%
Yeast	5	99.16%	39.1%	61.94%
Letter	7	100%	3.9%	12.3%

In this case, the performance related to the Wine, Vowel, and Letter benchmark problems remains the same as the initial DN design. With regard to the Yeast benchmark problem, there is

an improvement in the number of DNs used (only five) with a 1% difference in performance compared to the initial design. In general, this model performs better than the traditional network.

3.5.7 XOR & Shift Model

In this model, the DN is composed of 1 XOR unit and 1 Shift unit connected to 1 perceptron to provide the output of the DN as in the previous models. Half of the inputs are evaluated by the XOR unit and the remaining half of the inputs is evaluated by the Shift unit. Given an input vector pattern of length p, the mathematical model is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (x_{ik} \text{^} |w_{jk}|) \tag{3.13}$$

$$z_{j2} = \sum_{k=\frac{p}{2}}^{p-1} (\text{SH/NOP } (x_{ik,j}) + \text{ b} \tag{3.14}$$

$$Y_j = f[(z_{j1} * v_{j1}) + (z_{j2} * v_{j2}) + v_{j3}] \tag{3.15}$$

Where in the case of the XOR unit, x_{ik} represents the sign of the product of the k^{th} input of vector pattern i, and its corresponding weight w_{jk}, $|w_{jk}|$ is the magnitude of the weight corresponding to the link between input k and neuron j, and z_{j1} is the summation of the XOR operations between x_{ik} and $|w_{jk}|$ for the first half of the inputs. In the case of the Shift units, $x_{ik,j}$ represents the shifted (SH) or intact (NOP) input k of vector pattern i, connected to neuron j, z_{j2} is the summation of the remaining half of the shifted/intact inputs, b is a bias term, v_{j1} and v_{j2} are the internal weights of neuron j, v_{j3} is a bias term of neuron j, $f(x)$ is an activation function, and Y_j is the output of neuron j. The structure of this model is depicted in Figure 3.12 below:

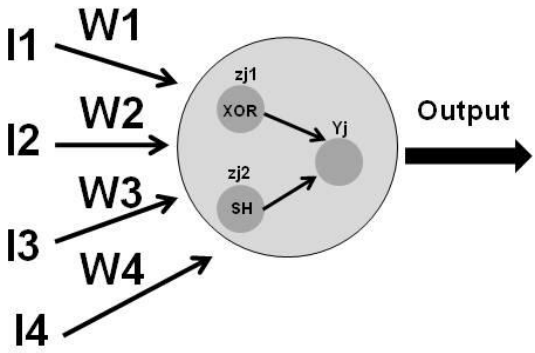


Figure 3.12 DN model with 1 XOR unit & 1 Shift unit

Comparing this model to the initial design, two activation functions are eliminated and the multiplication of the inputs is substituted by a simple XOR operation for the first half of the inputs and a simple shift operation for the remaining half of the inputs.

3.5.8 XOR & Shift Model Results

Using the bipolar sigmoid function, Table 3.71 shows the results of the network using DNs composed of 1 XOR unit and 1 Shift unit indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.71 Results of MLP with DNs Composed of 1 XOR Unit & 1 Shift Unit

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	0.0%	76.71%
Vowel	7	72.74%	9.1%	20.06%
Yeast	7	99.16%	39.1%	60.91%
Letter	7	61.35%	3.9%	11.84%

In this case, the performance related to the Wine benchmark problem remains the same as the initial DN design. With regard to the Yeast benchmark problem, only seven DNs provide a performance decreased by 1%. Finally, the performance related to the Vowel and Letter

benchmark problems decreased considerably compared to the initial design. In general, the performance of this model is lower than the traditional network's.

3.5.9 XOR & Quantum Model

In this model, the DN is composed of 1 XOR unit and 1 Quantum neuron connected to 1 perceptron. Half of the inputs are evaluated by the XOR unit and the remaining half of the inputs is evaluated by the Quantum neuron. Given an input vector pattern of length p, the mathematical model is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (x_{ik} \wedge |w_{jk}|) \tag{3.16}$$

$$z_{j2} = [1\ 1]^T . \sum_{k=p/2}^{p} x_{ik} | \emptyset_{\mathrm{jk}} > \tag{3.17}$$

$$Y_j = f[(z_{j1} * v_{j1}) + (z_{j2} * v_{j2}) + v_{j3}] \tag{3.18}$$

Where in the case of the XOR unit, x_{ik} represents the sign of the product of the k^{th} input of vector pattern i, and its corresponding weight w_{jk}, $|w_{jk}|$ is the magnitude of the weight corresponding to the link between input k and neuron j, and z_{j1} is the summation of the XOR operations between x_{ik} and $|w_{jk}|$ for the first half of the inputs. In the case of the quantum neuron, x_{ik} is the input k of vector pattern i, $\emptyset_{jk}$ is the weight coefficient k (converted to qbit) of the corresponding input k connected to DN j, z_{j2} is the evaluation of the remaining half of the inputs, v_{j1} and v_{j2} are the internal weights of neuron j, v_{j3} is a bias term of neuron j, $f(x)$ is an activation function, and Y_j is the output of neuron j. The structure of this model is depicted in Figure 3.13 below:

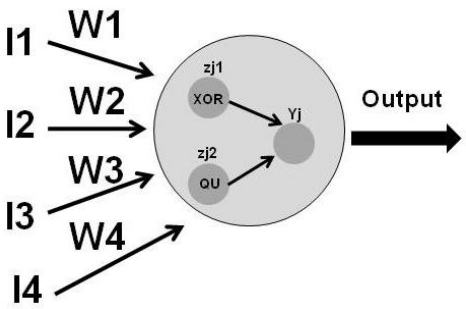


Figure 3.13 DN with 1 XOR unit & 1 Quantum neuron

Comparing this model to the initial design, two activation functions are eliminated and the multiplication of half of the inputs is substituted by a simple XOR operation and the size of the weight matrix increased 50% due to the quantum neuron.

3.5.10 XOR & Quantum Model Results

Using the bipolar sigmoid function, Table 3.72 shows the results of the network using DNs composed of 1 XOR unit and 1 Quantum neuron indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.72 Results of MLP with DNs Composed of 1 XOR Unit & 1 Quantum Neuron

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	0.0%	84.15%
Vowel	7	52.4%	9.1%	21.1%
Yeast	7	100%	39.1%	61.92%
Letter	8	100%	3.9%	13.8%

In this case, the performance related to the Wine, Yeast, and Letter benchmark problems remains the same as the initial DN design. However, for the Yeast benchmark problem, only seven DNs are required and for the Letter benchmark, one more DN is required. Finally, the performance

related to the Vowel benchmark problem dropped considerably compared to the initial design. In general, this model performs better than the traditional network.

3.5.11 Shift & Quantum Model

In this model, the DN is composed of 1 Shift unit and 1 Quantum neuron connected to 1 perceptron. Half of the inputs are evaluated by the Shift unit and the remaining half of the inputs is evaluated by the Quantum neuron. Given an input vector pattern of length *p*, the mathematical model is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (\text{SH/NOP } (x_{ik,j}) \tag{3.19}$$

$$z_{j2} = [1\ 1]^T . \sum_{k=p/2}^{p} x_{ik} | \emptyset_{\text{jk}} > \tag{3.20}$$

$$Y_j = f[(z_{j1} * v_{j1}) + (z_{j2} * v_{j2}) + v_{j3}] \tag{3.21}$$

Where in the case of the Shift unit, $x_{ik,\text{j}}$ represents the shifted (SH) or intact (NOP) input *k* of vector pattern *i*, connected to neuron *j*, and z_{j1} is the summation of the shifted/intact inputs for the first half of the inputs. In the case of the quantum neuron, x_{ik} is the input *k* of vector pattern *i*, $\emptyset_{jk}$ is the weight coefficient *k* (converted to qbit) of the corresponding input *k* connected to DN *j*, z_{j2} is the evaluation of the remaining half of the inputs, v_{j1} and v_{j2} are the internal weights of neuron *j*, v_{j3} is a bias term of neuron *j*, *f(x)* is an activation function, and Y_j is the output of neuron *j*. The structure of this model is depicted in Figure 3.14 below:

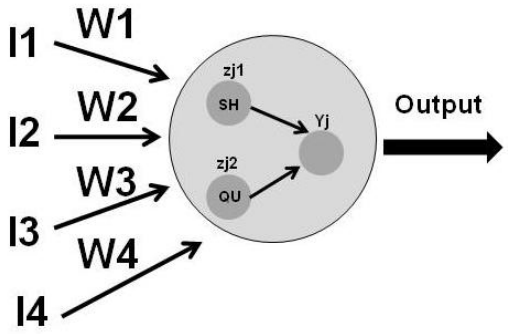


Figure 3.14 DN with 1 Shift unit & 1 Quantum neuron

Comparing this model to the initial design, two activation functions are eliminated and the multiplication of half of the inputs is substituted by a simple Shift operation and the size of the weight matrix increased 50% due to the quantum neuron.

3.5.12 Shift & Quantum Model Results

Using the bipolar sigmoid function, Table 3.73 shows the results of the network using DNs composed of 1 Shift unit and 1 Quantum neuron indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.73 Results of MLP with DNs Composed of 1 Shift Unit & 1 Quantum Neuron

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	0.0%	86.49%
Vowel	8	72.73%	9.1%	21.8%
Yeast	8	100%	39.1%	63.22%
Letter	8	100%	3.9%	13.55%

In this case, the performance related to the Wine, Yeast, and Letter benchmark problems remains the same as the initial DN design. However, for the Letter benchmark, one more DN is required. Finally, the performance related to the Vowel benchmark problem decreased significantly compared to the initial design. In general, this model performs better than traditional network.

3.5.13 Two XOR Units & 1 Quantum Neuron Model

In this model, the DN is composed of two XOR units connected to 1 Quantum neuron. Each half of the inputs is evaluated by one XOR unit and the output of the DN is generated through the quantum neuron. Given an input vector pattern of length *p*, the mathematical model is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (x_{ik} \wedge |w_{jk}|) \tag{3.22}$$

$$z_{j2} = \sum_{k=\frac{p}{2}}^{p-1} (x_{ik} \wedge |x_{jk}|) + \text{b} \tag{3.23}$$

$$Y_j = [1\ 1]^T . \sum_{k=0}^{2} z_{ik} | \emptyset_{\text{jk}} > \tag{3.24}$$

Where x_{ik} represents the sign of the product of the k^{th} input of vector pattern *i*, and its corresponding weight w_{jk}, $|w_{jk}|$ is the magnitude of the weight corresponding to the link between input *k* and neuron *j*, *b* is a bias term, z_{j1} is the summation of the XOR operations between x_{ik} and $|w_{jk}|$ for the first half of the inputs, z_{j2} is the summation of the XOR operations of the remaining half of the inputs, in the case of the quantum neuron (eq. 3.24), z_{ik} is the input *k* coming from the output of XOR unit *i*, $\emptyset_{jk}$ is the weight coefficient *k* (converted to qbit) of the corresponding input *k* connected to quantum neuron *j*, and Y_j is the output of DN neuron *j*. The structure of this model is depicted in Figure 3.15 below:

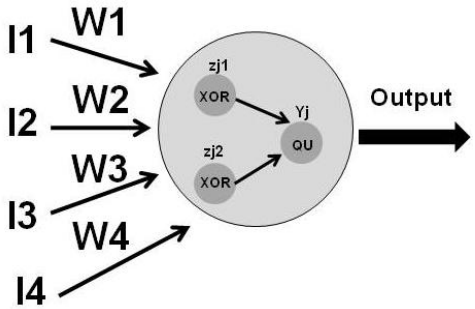


Figure 3.15 DN with 2 XOR units & 1 Quantum neuron

Comparing this model to the initial design, three activation functions are eliminated and the multiplication of the inputs is substituted by a simple XOR operation.

3.5.14 Two XOR Units & 1 Quantum Neuron Model Results

Using the bipolar sigmoid function, Table 3.74 shows the results of the network using DNs composed of 2 XOR units and 1 Quantum neuron indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.74 Results of MLP with DNs Composed of 2 XOR Units & 1 Quantum Neuron

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	0.0%	35.23%
Vowel	7	95.7%	10.61%	41.87%
Yeast	7	100%	39.1%	55.28%
Letter	7	100%	7.28%	23.88%

In this case, the performance related to the Wine, Yeast, and Letter benchmark problems remains the same as the initial DN design. However, for the Yeast benchmark problem, only seven DNs are required. Finally, the performance of the Vowel benchmark problem decreased only 4.3%. In general, this model performs better than the traditional network.

3.5.15 Two Shift Units & 1 Quantum Neuron Model

In this model, the DN is composed of two Shift units connected to 1 Quantum neuron. Each half of the inputs is evaluated by one Shift unit and the output of the DN is generated through the quantum neuron. Given an input vector pattern of length p, the mathematical model is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (\text{SH/NOP}\ (x_{ik,j}) \tag{3.25}$$

$$z_{j2} = \sum_{k=\frac{p}{2}}^{p-1} (\text{SH/NOP}\ (x_{ik,j}) + \text{b} \tag{3.26}$$

$$Y_j = [1\ 1]^T . \sum_{k=0}^{2} z_{ik} | \emptyset_{\text{jk}} > \tag{3.27}$$

Where $x_{ik,j}$ represents the shifted (SH) or intact (NOP) input k of vector pattern i, connected to neuron j, z_{j1} is the summation of the shifted/intact inputs for the first half of the inputs, z_{j2} is the summation of the remaining half of the shifted/intact inputs, in the case of the quantum neuron (eq. 3.24), z_{ik} is the input k coming from the output of shift unit i, $\emptyset_{jk}$ is the weight coefficient k (converted to qbit) of the corresponding input k connected to quantum neuron j, and Y_j is the output of DN neuron j. The structure of this model is depicted in Figure 3.16 below:

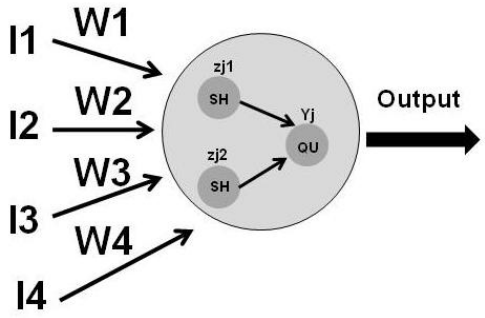


Figure 3.16 DN with 2 Shift units & 1 Quantum neuron

Comparing this model to the initial design, three activation functions are eliminated and the multiplication of the inputs is substituted by a simple shift operation.

3.5.16 Two Shift Units & 1 Quantum Neuron Model Results

Using the bipolar sigmoid function, Table 3.75 shows the results of the network using DNs composed of 2 Shift units and 1 Quantum neuron indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.75 Results of MLP with DNs Composed of 2 Shift Units & 1 Quantum Neuron

Dataset	#DNs	Max	Min	Avg
Wine	8	100%	100%	100%
Vowel	7	100%	10.61%	43.47%
Yeast	7	100%	39.1%	60.10%
Letter	7	100%	7.28%	23.88%

In this case, the performance is the same as the initial DN design. However, for the Vowel and Yeast benchmark problems, only seven DNs are required. In addition, the Wine benchmark problem required more DNs (8). In general, this performs better than the traditional network.

3.5.17 Two Quantum Neurons & 1 XOR Unit Model

In this model, the DN is composed of two quantum neurons connected to 1 XOR unit. Each half of the inputs is evaluated by one quantum neuron and the output of the DN is generated through the XOR unit. Given an input vector pattern of length *p*, the mathematical model is described below:

$$z_{j1} = [1\ 1]^T . \sum_{k=0}^{\left(\frac{p}{2}\right)-1} x_{ik} | \emptyset_{\mathrm{jk}} > \tag{3.28}$$

$$z_{j2} = [1\ 1]^T . \sum_{k=p/2}^{p} x_{ik} | \emptyset_{\mathrm{jk}} > \tag{3.29}$$

$$Y_j = \sum_{k=0}^{2} (z_{ik} \hat{} \ |w_{jk}|) \tag{3.30}$$

Where x_{ik} is the input k of vector pattern i, $\emptyset_{jk}$ is the weight coefficient k (converted to qbit) of the corresponding input k connected to DN j, z_{j1} is evaluation of the first half of the inputs, z_{j2} is the evaluation of the remaining half of the inputs, z_{ik} represents the inputs connected to the quantum neuron, $|w_{jk}|$ is the magnitude of the weight coefficient corresponding to the input z_{ik}, and Y_j is the output of DN j. The structure of this model is depicted in Figure 3.17 below:

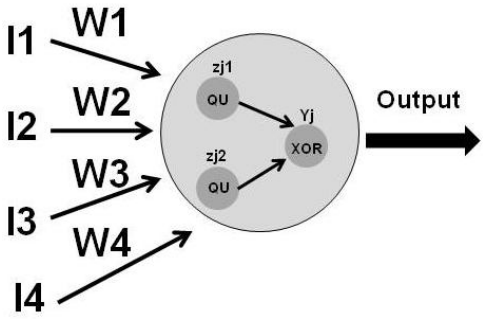


Figure 3.17 DN with 2 Quantum neurons & 1 XOR unit

Comparing this model to the initial design, three activation functions are eliminated and the size of the weight matrix doubled.

3.5.18 Two Quantum Neurons & 1 XOR Unit Model Results

Using the bipolar sigmoid function, Table 3.76 shows the results of the network using DNs composed of two quantum units and one XOR unit indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.76 Results of MLP with DNs Composed of 2 Quantum Neurons & 1 XOR Unit

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	0.0%	79.61%
Vowel	7	72.73%	9.1%	18.84%
Yeast	8	99.16%	39.1%	64.94%
Letter	7	100%	17.54%	3.9%

In this case, the performance of the Wine and Letter benchmark problems is the same as the initial DN design. However, for the Yeast benchmark problem, the performance decreased about 1%. In addition, regarding the Vowel benchmark problem, the performance dropped significantly. In general, this model performs better than the traditional network.

3.5.19 Two Quantum Neurons & 1 Shift Unit Model

In this model, the DN is composed of two quantum neurons connected to 1 Shift unit. Each half of the inputs is evaluated by one quantum neuron and the output of the DN is generated through the Shift unit. Given an input vector pattern of length *p*, the mathematical model is described below:

$$z_{j1} = [1\ 1]^T . \sum_{k=0}^{\left(\frac{p}{2}\right)-1} x_{ik} | \emptyset_{\mathrm{jk}} > \tag{3.31}$$

$$z_{j2} = [1\ 1]^T . \sum_{k=p/2}^{p} x_{ik} | \emptyset_{\mathrm{jk}} > \tag{3.32}$$

$$Y_j = \sum_{k=0}^{1} (\mathrm{SH/NOP}\ (z_{ik,j}) + \mathrm{b} \tag{3.33}$$

Where x_{ik} is the input *k* of vector pattern *i*, $\emptyset_{jk}$ is the weight coefficient *k* (converted to qbit) of the corresponding input *k* connected to DN *j*, z_{j1} is evaluation of the first half of the inputs, z_{j2} is the evaluation of the remaining half of the inputs, $z_{ik,j}$ represents the inputs connected to the shift unit, b is a bias term, and Y_j is the output of DN *j*. The structure of this model is depicted in Figure 3.18 below:

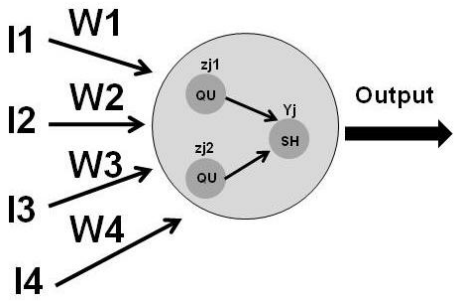


Figure 3.18 DN with 2 Quantum neurons & 1 Shift unit

Comparing this model to the initial design, three activation functions are eliminated and the size of the weight matrix doubled.

3.5.20 Two Quantum Neurons & 1 Shift Unit Model Results

Using the bipolar sigmoid function, Table 3.77 shows the results of the network using DNs composed of two quantum units and one Shift unit indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.77 Results of MLP with DNs Composed of 2 Quantum Neurons & 1 Shift Unit

Dataset	#DNs	Max	Min	Avg
Wine	8	100%	100%	100%
Vowel	7	100%	9.1%	44%
Yeast	7	89.1%	39.1%	48.81%
Letter	8	61.63%	3.9%	18.19%

In this case, the performance of the Wine and Vowel benchmark problems is the same as the initial DN design. However, for the Yeast and Vowel benchmark problems, the performance decreased. In general, the performance of this model is lower than the traditional network.

3.5.21 3-XOR Units Model

In this model, the DN is composed of three XOR units. Each half of the inputs is evaluated by one of the first two XOR units and the output of the DN is generated through the third XOR unit. Given an input vector pattern of length p, the mathematical model is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (x_{ik} \wedge |w_{jk}|) \tag{3.34}$$

$$z_{j2} = \sum_{k=\frac{p}{2}}^{p-1} (x_{ik} \wedge |x_{jk}|) + \text{b} \tag{3.35}$$

$$Y_j = \sum_{k=0}^{2} (z_{ik} \wedge |w_{jk}|) \tag{3.36}$$

Where x_{ik} represents the sign of the product of the k^{th} input of vector pattern i, and its corresponding weight w_{jk}, $|w_{jk}|$ is the magnitude of the weight corresponding to the link between

input k and neuron j, b is a bias term, z_{j1} is the summation of the XOR operations between x_{ik} and $|w_{jk}|$ for the first half of the inputs, z_{j2} is the summation of the XOR operations of the remaining half of the inputs, and Y_j is the output of DN j provided by the third XOR unit. The structure of this model is depicted in Figure 3.19 below:

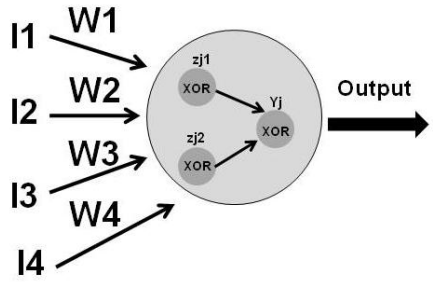


Figure 3.19 DN with 3 XOR units

Comparing this model to the initial design, three activation functions are eliminated and the multiplication is substituted by a simple XOR operation.

3.5.22 3-XOR Units Model Results

Using the bipolar sigmoid function, Table 3.78 shows the results of the network using DNs composed of three XOR units indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.78 Results of MLP with DNs Composed of 3 XOR Units

Dataset	#DNs	Max	Min	Avg
Wine	5	100%	0.0%	81.01%
Vowel	8	72.73%	9.1%	23.43%
Yeast	9	100%	5.72%	44.15%
Letter	9	100%	3.9%	16.15%

In this case, the performance of the Wine, Yeast, and Letter benchmark problems is the same as the initial DN design. However, with regard to the Vowel benchmark problem, the performance decreased significantly. In general, the performance of this model is better than the traditional network.

3.5.23 3-Shift Units Model

In this model, the DN is composed of three Shift units. Each half of the inputs is evaluated by one of the first two Shift units and the output of the DN is generated through the third Shift unit. Given an input vector pattern of length p, the mathematical model is described below:

$$z_{j1} = \sum_{k=0}^{\left(\frac{p}{2}\right)-1} (\text{SH/NOP}\ (x_{ik,j}) \tag{3.37}$$

$$z_{j2} = \sum_{k=\frac{p}{2}}^{p-1} (\text{SH/NOP}\ (x_{ik,j}) + \text{b} \tag{3.38}$$

$$Y_j = \sum_{k=0}^{1} (\text{SH/NOP}\ (z_{ik,j}) + \text{b} \tag{3.39}$$

Where $x_{ik,j}$ represents the shifted (SH) or intact (NOP) input k of vector pattern i, connected to neuron j based on the rules previously described. z_{j1} is the summation of the first half of the shifted/intact inputs, z_{j2} is the summation of the remaining half of the shifted/intact inputs, b is a bias term, $z_{ik,j}$ is the output of the first two shift units connected to the third shift unit, and Y_j is the output of DN j provided by the third Shift unit. The structure of this model is depicted in Figure 3.20 below:

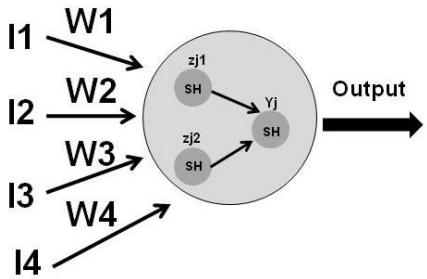


Figure 3.20 DN with 3 Shift units

Comparing this model to the initial design, three activation functions are eliminated and the multiplication is substituted by a simple Shift operation.

3.5.24 3-Shift Units Model Results

Using the bipolar sigmoid function, Table 3.79 shows the results of the network using DNs composed of three Shift units indicating the number of DNs in the hidden layer followed by the corresponding performance:

Table 3.79 Results of MLP with DNs Composed of 3 Shift Units

Dataset	#DNs	Max	Min	Avg
Wine	5	40.54%	40.54%	40.54%
Vowel	7	54.3%	13.42%	23.97%
Yeast	8	89.1%	68.69%	83.6%
Letter	7	54%	54%	54%

In this case, the performance of the Wine, Vowel, Yeast, and Letter benchmark problems is significantly lower than the both the initial DN design and the traditional network.

3.6 Performance and Simulation Time Experiment Results

The performance of each of the models proposed and the traditional network were evaluated and ranked starting with the best performance. In addition, the simulation time of each of the models, including the traditional network, was recorded and ranked to observe the

computational load provided by each of the models presented. Table 3.80 shows the ranking of the performance and simulation time of the experiment results obtained by listing the neuron model used (M), the activation function used (F(x)), the benchmark problem (Bench), number of hidden neurons (H-Ns), the corresponding performance (Max, Avg, and Min), the performance ranking (P-R), the corresponding simulation time (ST), and the simulation time ranking (ST-R).

The main goal of this research is to observe if the DN model increases the performance of the traditional network. Consequently, based on the experiment results obtained, there are ten Divcon Neuron models that performed better than the traditional network, which are the *initial design*, the *XOR Units model*, the *Shift Units model*, the *Quantum Neurons model*, the *XOR Unit-Quantum Neuron model*, the *Shift Unit-Quantum Neuron model*, the *two XOR Units-1Quantum Neuron model*, the *two Shift Units-1 Quantum Neuron model*, the *two Quantum Neurons-1 XOR Unit model*, and the *three XOR Units model*. However, there are three Divcon Neuron models that performed lower than the traditional network. These are the *XOR Unit-Shift Unit model*, the *two Quantum Neurons-1 Shift Unit model*, and the *three Shift Units model*.

3.6.1 Performance Experiment Results Evaluation

As indicated in Table 3.80, the best performance was provided by the *two Shift Units-1 Quantum Neuron model*. This DN model composed of two shift units connected to one quantum neuron provided the best performance of all the different models developed, that is, 100% recognition performance for all benchmark problems tested. In addition, it requires less DNs in the hidden layer for the large datasets. However, it is important to mention that there is a very small difference between the performances of the first four best models.

Table 3.80 Performance and Simulation Time Experiment Results

M	F(x)	Bench	H-Ns	Max	Avg	Min	P-R	ST	ST-R
Crisp	Bip-Sig	Wine	22	100%	61.86%	0.0%	10	7m-34s	14
		Vowel		100%	27.02%	9.1%		44m-17s	
		Yeast		99.16%	61.08%	39.1%		70m-48s	
		Letter		61.63%	20.1%	3.9%		45h-1m-59s	
Intial	Bip-Sig	Wine	5	100%	77.04%	0.0%	2	4m-39s	1
		Vowel	8	100%	24.08%	9.1%		26m-35s	
		Yeast	8	100%	57.1%	39.1%		43m-1s	
		Letter	7	100%	15.13%	3.9%		17h-27m-10s	
XOR	Bip-Sig	Wine	5	100%	81.28%	0.0%	6	6m-6s	4
		Vowel	7	54.11%	15.53%	9.1%		22m-2s	
		Yeast	5	100%	61.13%	39.1%		23m-14s	
		Letter	7	100%	12.1%	3.9%		21h-14m	
Shift	Bip-Sig	Wine	5	100%	74.84%	0.0%	8	3m-50s	13
		Vowel	11	98.92%	22.56%	9.1%		33m-25s	
		Yeast	5	99.16%	63.41%	39.1%		27m-10s	
		Letter	10	100%	14.6%	3.9%		31h-14m-37s	
Quant	Bip-Sig	Wine	5	100%	92.99%	32.43%	3	3m	8
		Vowel	8	100%	17.33%	9.1%		35m-3s	
		Yeast	5	99.16%	61.94%	39.1%		27m-25s	
		Letter	7	100%	12.3%	3.9%		25h-10m-29s	
XO-SH	Bip-Sig	Wine	5	100%	76.71%	0.0%	12	3m-9s	3
		Vowel	7	72.73%	20.06%	9.1%		21m-5s	
		Yeast	7	99.16%	60.91%	39.1%		32m-42s	
		Letter	7	61.35%	11.84%	3.9%		20h-52m-49s	
XO-QU	Bip-Sig	Wine	5	100%	84.15%	0.0%	7	4m-23s	12
		Vowel	7	52.4%	21.1%	9.1%		31m-6s	
		Yeast	7	100%	61.92%	39.1%		46m-48s	
		Letter	8	100%	13.8%	3.9%		29h-31m-43s	
SH-QU	Bip-Sig	Wine	5	100%	86.49%	0.0%	5	4m-2s	11
		Vowel	8	72.73%	21.8%	9.1%		28m-49s	
		Yeast	8	100%	63.22%	39.1%		43m-16s	
		Letter	8	100%	13.55%	3.9%		28h-15m-42s	
2X-QU	Bip-Sig	Wine	5	100%	35.23%	0.0%	4	9m-46s	7
		Vowel	7	95.7%	41.87%	10.61%		38m-14s	
		Yeast	7	100%	55.28%	39.1%		54m-33s	
		Letter	7	100%	23.88%	7.28%		25h-30s	
2S-QU	Bip-Sig	Wine	8	100%	100%	100%	1	4m-13s	9
		Vowel	7	100%	43.47%	10.61%		39m-5s	
		Yeast	7	100%	60.1%	39.1%		53m-42s	
		Letter	7	100%	23.88%	7.28%		27h-4m-23s	
2Q-1X	Bip-Sig	Wine	5	100%	79.61%	0.0%	9	3m-56s	5
		Vowel	7	72.73%	18.84%	9.1%		27m-16s	
		Yeast	8	99.16%	64.94%	39.1%		37m-31s	
		Letter	7	100%	17.54%	3.9%		21h-47m-55s	
2Q-1S	Bip-Sig	Wine	8	100%	100%	100%	11	4m	10
		Vowel	7	100%	44%	9.1%		43m-39s	
		Yeast	7	89.1%	48.81%	39.1%		54m-51s	
		Letter	8	61.63%	18.19%	3.9%		28h-15m-7s	
3-XOR	Bip-Sig	Wine	5	100%	81.01%	0.0%	5	4m-41s	2
		Vowel	8	72.73%	23.43%	9.1%		30m-28s	
		Yeast	9	100%	44.15%	5.72%		47m-28s	
		Letter	9	100%	16.15%	3.9%		19h-44m-57s	
3-SH	Bip-Sig	Wine	5	40.54%	40.54%	40.54%	13	2m-51s	6
		Vowel	7	54.3%	23.97%	13.42%		27m-8s	
		Yeast	8	89.1%	83.6%	68.69%		59m-42s	
		Letter	7	54%	54%	54%		23h-5m-3s	

In addition, three out of the first four best models use quantum neurons, which provide an indication that quantum neurons make a good contribution to increasing the processing power of the DN model. Moreover, taking into account that ten of the proposed DN models performed better than the traditional network, these results demonstrate that the concept proposed by the DN models successfully contributes to increase the performance of a traditional feed forward MLP.

3.6.2 Simulation Time Experiment Results Evaluation

One of the additional aspects of this research to be observed is the simulation time, as table 3.80 shows, the traditional network provides the longest simulation time. Based on the results, all DN models provide better simulation time, which means that for the benchmark problems selected, the computational load was reduced. This reduction in computational load provides an additional benefit, which is always good to consider when a particular neural network model is to be selected.

Chapter 4

HARDWARE IMPLEMENTATION OF DN MODELS

4.1 Hardware Implementation

The second aspect of this research to be observed is the hardware implementation of the traditional network and the DN models proposed to compare the resources utilized. A good alternative for a hardware implementation, composed of individual ICs, is the use of an FPGA, which has been used for many implementations of ANNs [ALK08, CHU08, DAW09, GOK05, GUP09, HAU08, JOY07, LEI08, LOR08, MOR09, NOO03, ORL11, RIC09, SEO03, and TUK10]. The different DN models and the traditional neuron models were implemented using the commonly used hardware description language called VHDL [BAI08, BRO09, CHU08, EIC06, HAU08, JOY07, and YAL05]. These models were implemented using a Xilinx Spartan 3 FPGA model xc3s50-4pq208 under Xilinx ISE 10.1 environment. All the neuron models were designed to process 8-bit signed integers as inputs and provide an 8-bit binary output.

4.2 Traditional Perceptron Models Hardware Implementation

The design of the traditional perceptron model consists of two designs. The first design implements the perceptron to be used in the hidden layer and the second design implements the perceptron to be used in the output layer. Figure 4.1 shows the design of the hidden layer perceptron. This is a typical design that stores the weight coefficients in RAM obtained by the offline training of the ANN. Once the weights are stored in RAM, and then the inputs are fed to the perceptron serially and multiplied by their corresponding weight coefficient (note that the serial strategy is typical, because a fully parallel implementation is not general and very

resource-intensive). All 16-bit products are summed and the resulting accumulated sum is fed to the activation function F(x), which provides the output of the perceptron.

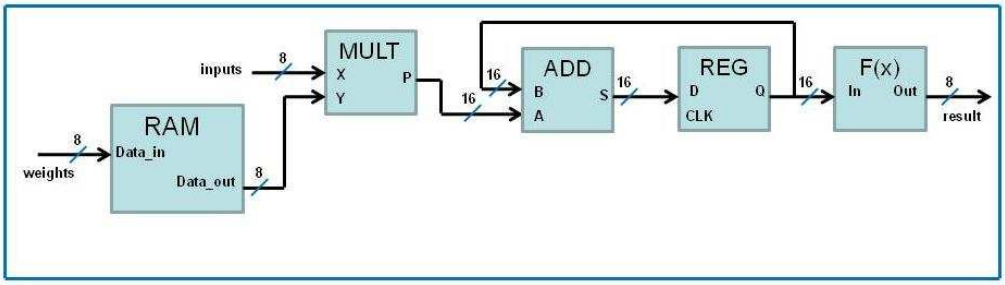


Figure 4.1 Hidden Layer Perceptron Hardware Design

Figure 4.2 shows the design of the output layer perceptron, which is basically the same design, but the activation function is substituted with a comparator to generate a binary output.

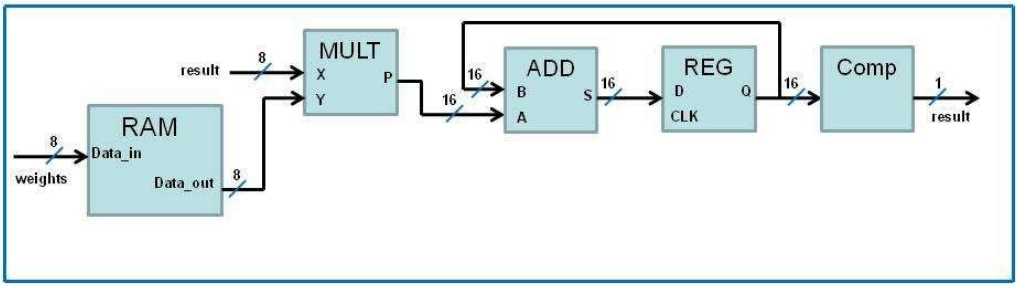


Figure 4.2 Output Layer Perceptron Hardware Design

4.3 DN Models Hardware Implementation

Following the same approach, the DN models were implemented to observe the hardware resources utilized compared to the traditional network.

4.3.1 Initial DN Model Hardware Implementation

The initial Divcon Neuron model design is depicted in Figure 4.3, showing the additional computational power embedded in this neuron model:

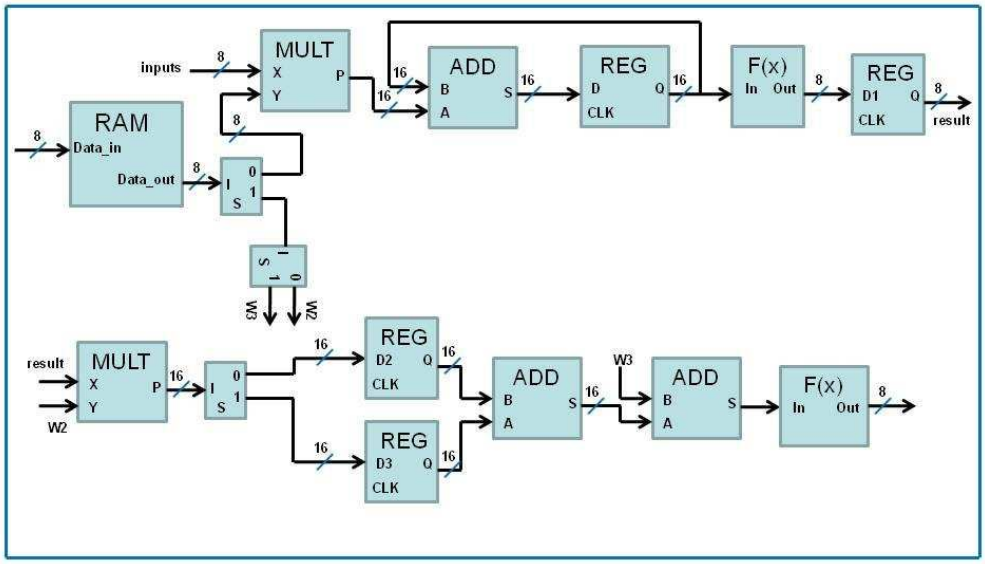


Figure 4.3 Initial DN Model Hardware Design

This design processes the first half of the inputs and the result is stored in register *D1*, then this result is sent through the signal "*result*" to be multiplied by its corresponding inner weight and the result is stored register *D2*. In the mean time, the second half of the inputs is processed in the same manner and the result is again sent through the signal "*result*" to be processed and this result is stored in register *D3*. After this, D2, D3, and a bias term *W3* are added and the result is fed to the activation function *F(x)* to generate the output of the DN.

4.3.2 XOR Units Model Hardware Implementation

The XOR Units model design is shown in Figure 4.4. The XOR unit processes the first half of the inputs (serially) and the resulting accumulated sum is stored in register *D1*, which sends this sum through the signal "result" to be processed and the result is stored in register *D2*. In the mean time, the second half of the inputs is processed in the same manner and the result is stored in register *D3*. Finally, the results from *D2* and *D3* are added to a bias term *W3* and the result is sent to the activation function *F(x)* to generate the output of the DN.

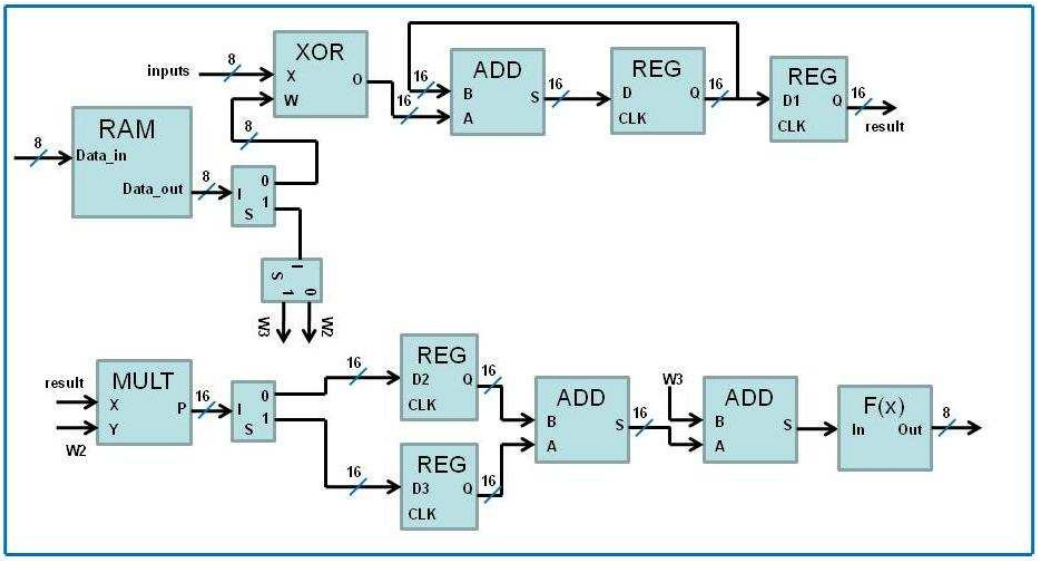


Figure 4.4 XOR Units Model Hardware Design

4.3.3 Shift Units Model Hardware Implementation

The Shift Units model design is very similar to the XOR Units design and is depicted in Figure 4.5 below:

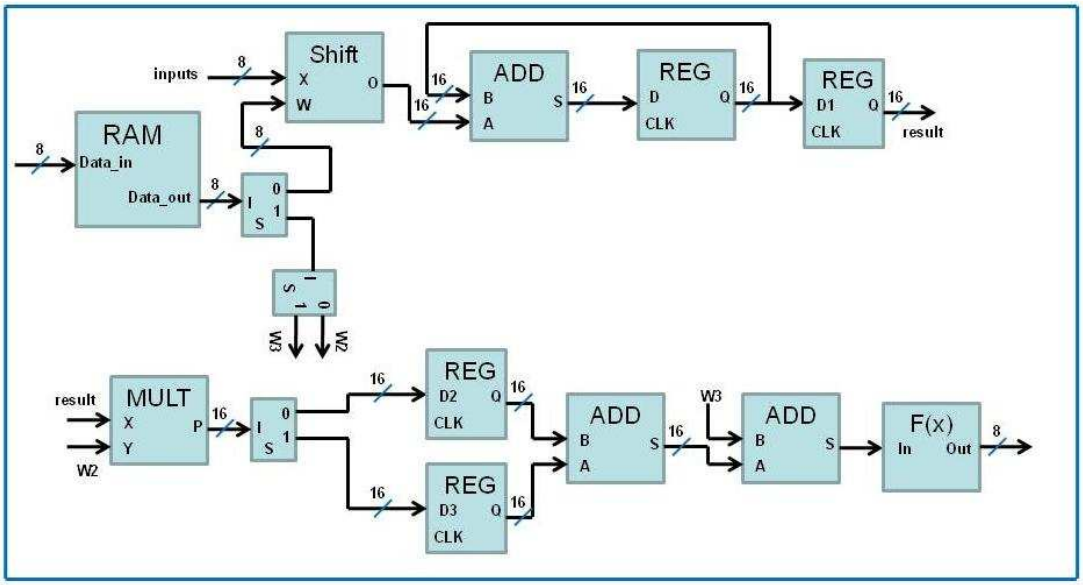


Figure 4.5 Shift Units Model Hardware Design

The Shift unit processes the first half of the inputs, which are fed serially to the shift unit. The resulting accumulated sum is stored in register *D1*, which sends this sum through the signal "result" to be processed and the result is stored in register *D2*. In the mean time, the second half of the inputs is processed in the same manner and the result is stored in register *D3*. Finally, the results from *D2* and *D3* are added to a bias term *W3* and the result is sent to the activation function *F(x)* to generate the output of the DN.

4.3.4 Quantum Neurons Model Hardware Implementation

The design of the quantum neurons model is shown in Figure 4.6 below:

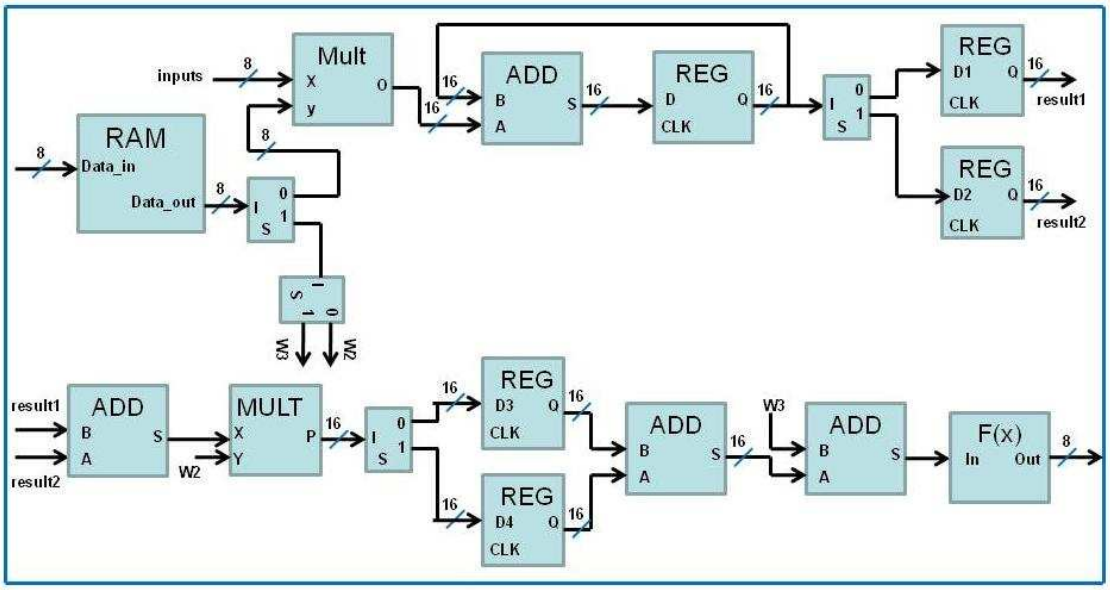


Figure 4.6 Quantum Neurons Model Hardware Design

In this case, the first half of the inputs is evaluated with the first set of qbit weights and the accumulated sum is stored in register *D1*. Then the same inputs are evaluated with the second set of qbit weights and the accumulated sum is stored in register *D2*. Then these results are sent to be added through signals "*result1*" and "*result2*" producing the output of a quantum neuron.

The result of this sum, for the first half of the inputs is stored in register *D3*. The same process is repeated for the remaining half of the inputs and the result is stored in register *D4*. Finally, these two results are added including a bias term *W3* and the result is fed to the activation function *F(x)* to generate the output of the DN.

4.3.5 XOR & Shift Model Hardware Implementation

Figure 4.7 depicts the XOR & Shift Model design below:

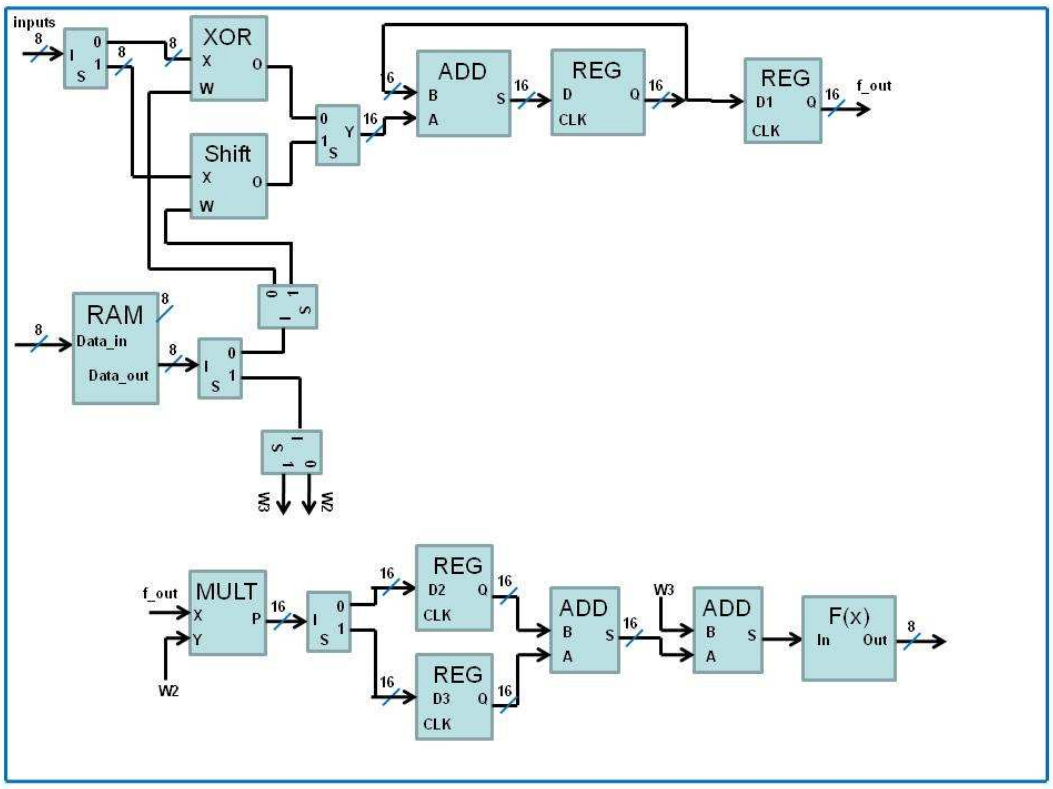


Figure 4.7 XOR & Shift Model Hardware Design

The first half of the inputs is evaluated serially through the XOR unit, and the resulting accumulated sum is stored in register *D1*. Then this sum is sent to be processed through the signal "*f_out*" and the result is stored in register *D2*. In the mean time, the second half of the inputs is evaluated through the Shift unit, and the resulting accumulated sum is stored in register

D1. Then again, this sum is sent to be processed and the result is stored in register D3. Finally, the results of *D2* and *D3* are added including a bias term *W3* and result is fed to the activation function *F(x)* which provides the output of the DN.

4.3.6 XOR & Quantum Model Hardware Implementation

The design of the XOR & Quantum model is shown in Figure 4.8 below:

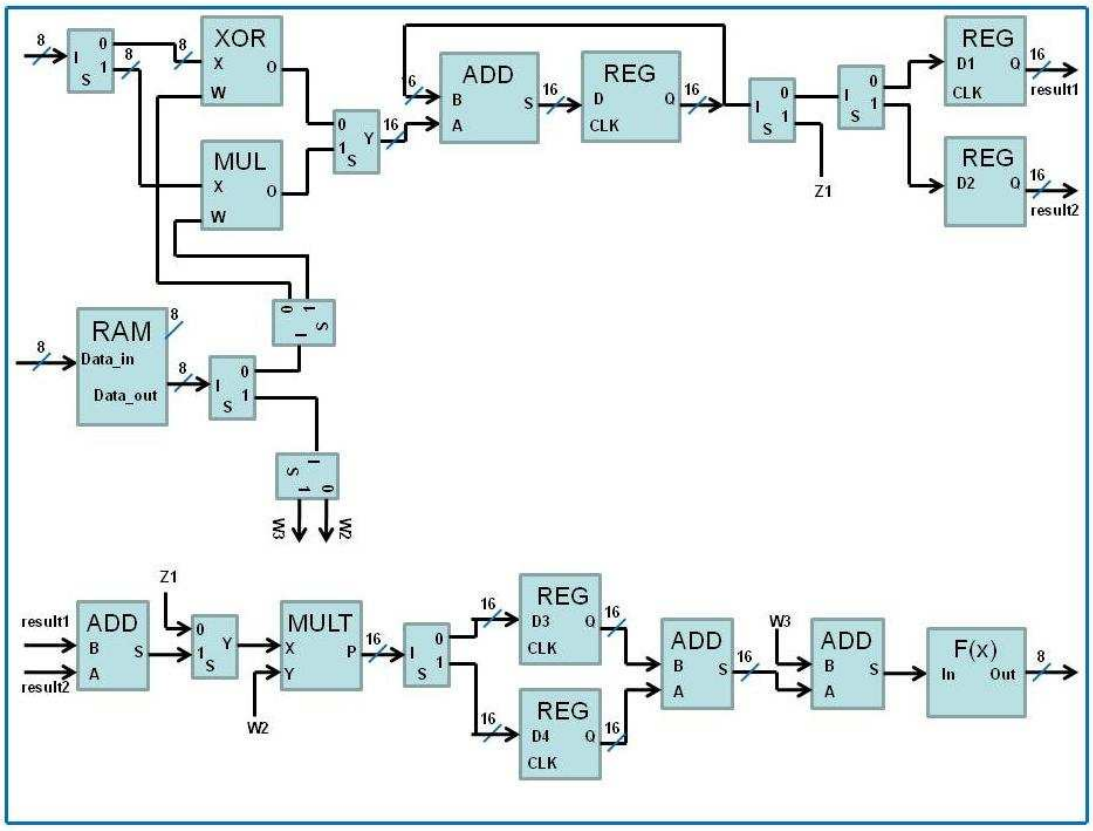


Figure 4.8 XOR & Quantum Model Hardware Design

The first half of the inputs is processed through the XOR unit and the accumulated sum is sent to be processed through signal "*Z1*" and the result is stored in register *D3*. The second half of the inputs is processed by a quantum neuron, so the inputs go through the multiplier and the accumulated sum due to the first set of qbit weights is stored in register *D1*. Then the process is

repeated for the second set of qbit weights and the accumulated sum is stored in register *D2*. Then the results from *D1* and *D2* are sent through the signals "*result1*" and "*result2*" to be processed and the result is stored in register *D4*. Finally *D3* and *D4* are added including a bias term *W3* and the sum is sent to the activation function *F(x)* to generate the output of the DN.

4.3.7 Shift & Quantum Model Hardware Implementation

The shift & quantum model design is very similar to the XOR & quantum design and is shown in Figure 4.9:

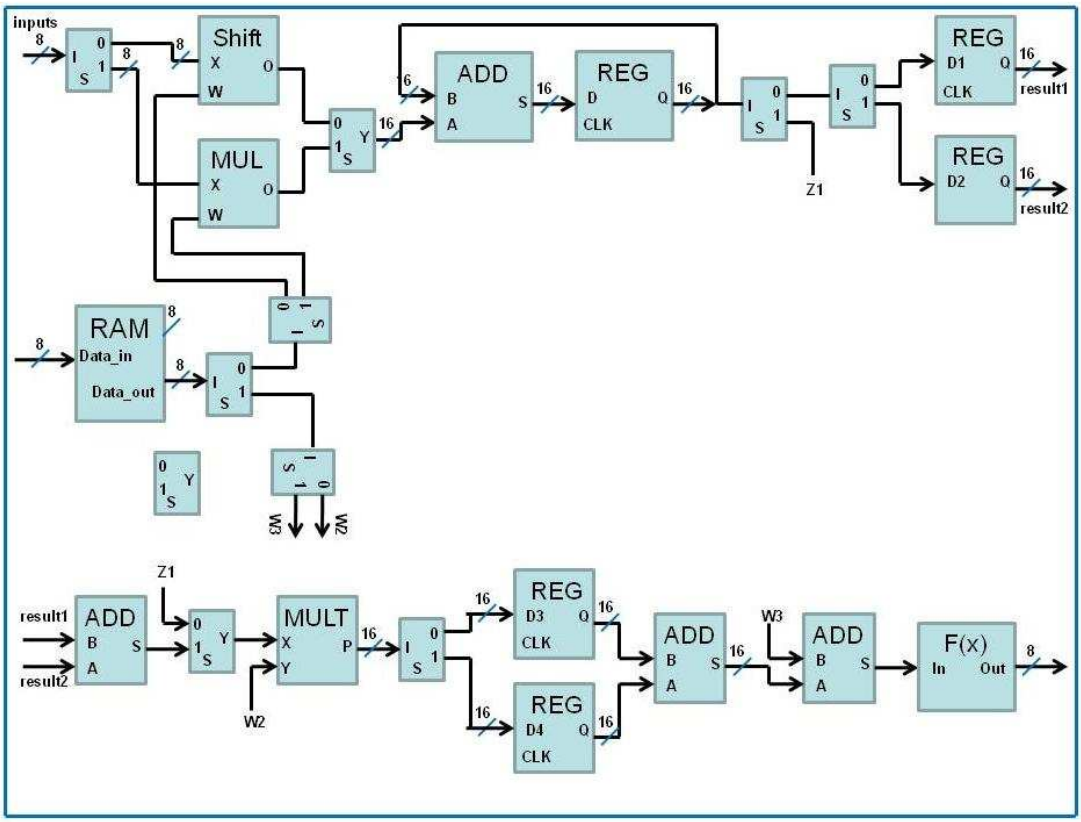


Figure 4.9 Shift & Quantum Model Hardware Design

The first half of the inputs is processed through the Shift unit and the accumulated sum is sent to be processed through signal "*Z1*" and the result is stored in register *D3*. The second half of the

inputs is processed by a quantum neuron, so the inputs go through the multiplier and the accumulated sum due to the first set of qbit weights is stored in register *D1*. Then the process is repeated for the second set of qbit weights and the accumulated sum is stored in register *D2*. Then the results from *D1* and *D2* are sent through the signals "*result1*" and "*result2*" to be processed and the result is stored in register *D4*. Finally *D3* and *D4* are added including a bias term *W3* and the sum is sent to the activation function *F(x)* to generate the output of the DN.

4.3.8 2-XOR Units & 1 Quantum Neuron Model Hardware Implementation

The two XOR units & 1 quantum neuron model design is depicted in Figure 4.10 below:

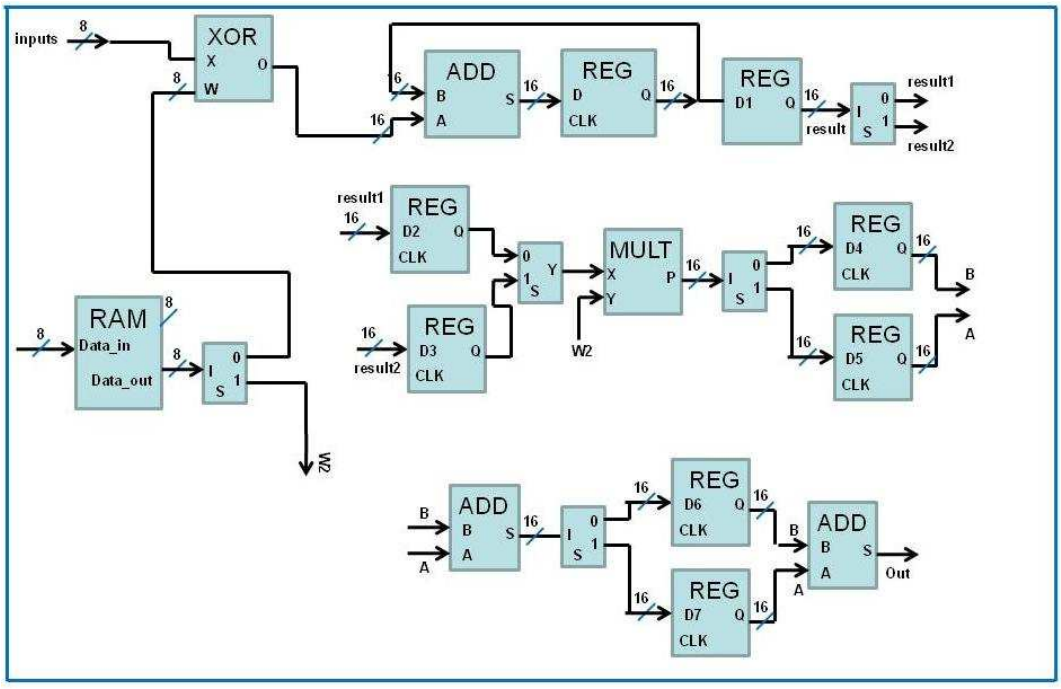


Figure 4.10 2-XOR Units & 1 Quantum Neuron Model Hardware Design

In this case, the first half of the inputs is evaluated through the XOR unit and the accumulated sum is stored in register *D1*. Then through signal "*result1*" the sum is stored in register *D2*. In the mean time, the same process is repeated for the second half of the inputs and the sum is

stored in register *D3*. Then *D2* is multiplied times the qbit weight *W2* and the corresponding results are stored in registers *D4* and *D5* respectively. Then *D4* and *D5* are added and the sum is stored in register *D6*. The same process is repeated using *D3* and the result is stored in register *D7*. Finally, D6 and D7 are added and the result provides the output of the DN.

4.3.9 2-Shift Units & 1 Quantum Neuron Model Hardware Implementation

This design is very similar to the previous design described, which is shown in Figure 4.11 below:

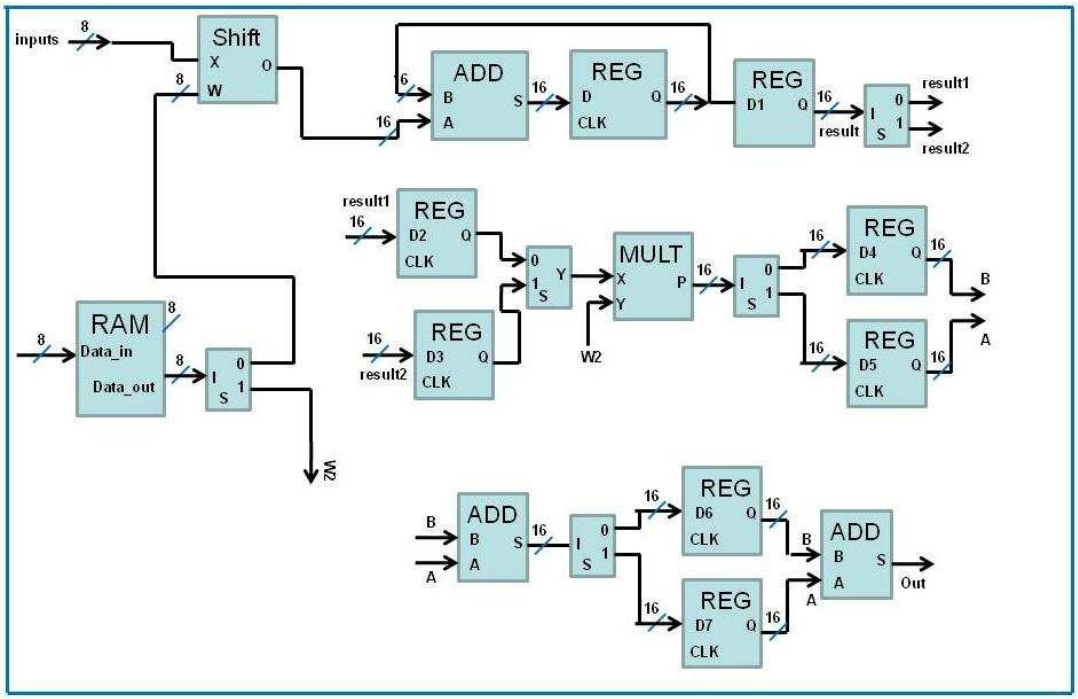


Figure 4.11 2-Shift Units & 1 Quantum Neuron Model Hardware Design

In this case, the first half of the inputs is evaluated through the Shift unit and the accumulated sum is stored in register *D1*. Then through signal "*result1*" the sum is stored in register *D2*. In the mean time, the same process is repeated for the second half of the inputs and the sum is stored in register *D3*. Then *D2* is multiplied times the qbit weight *W2* and the corresponding

results are stored in registers *D4* and *D5* respectively. Then *D4* and *D5* are added and the sum is stored in register *D6*. The same process is repeated using *D3* and the result is stored in register *D7*. Finally, D6 and D7 are added and the result provides the output of the DN.

4.3.10 2-Quantum Neurons & 1 XOR Unit Model Hardware Implementation

Figure 4.12 depicts the two-quantum neurons and one XOR unit model design:

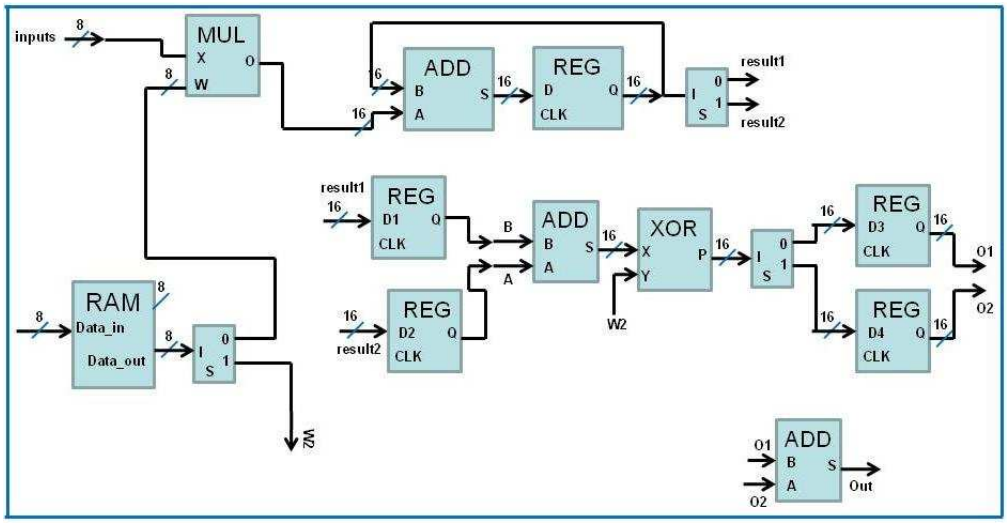


Figure 4.12 2-Quantum Neurons & 1 XOR Unit Model Hardware Design

For this design, the first half of the inputs is evaluated and the resulting accumulated sum related to the first set of qbit weights is stored in register *D1*. The accumulated sum related to the second set of qbit weights is stored in register *D2*. *D1* and *D2* are added providing the output of the quantum neuron to the *XOR* unit. Once evaluated by the *XOR* unit, the result is stored in register *D3*. In the mean time, the remaining half of the inputs is evaluated following the same procedure and the result is stored in register *D4*. Finally, *D4* and *D3* are added and the result is the output of the DN.

4.3.11 2-Quantum Neurons & 1 Shift Unit Model Hardware Implementation

The two-quantum neurons and one shift unit model design is very similar to the previous implementation and is shown in Figure 4.13:

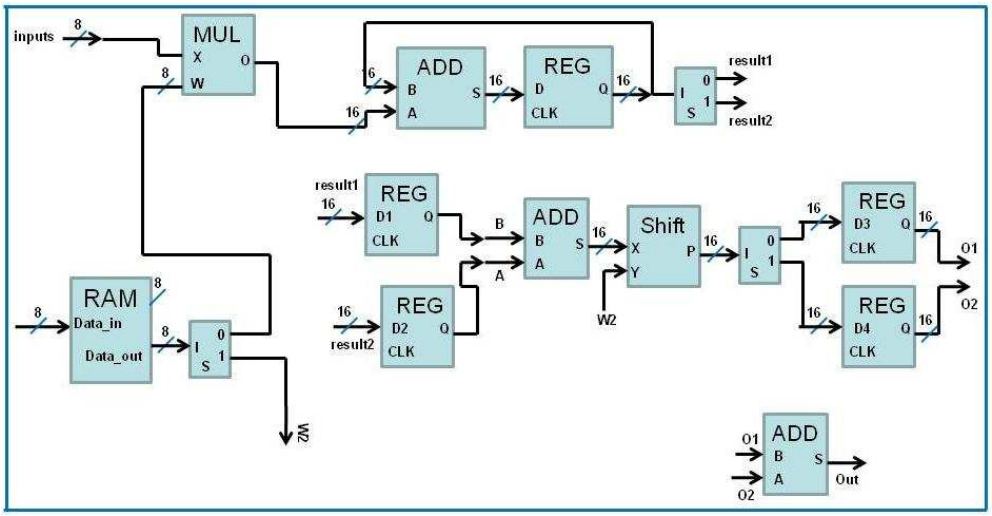


Figure 4.13 2-Quantum Neurons & 1 Shift Unit Model Hardware Design

For this design, the first half of the inputs is evaluated and the resulting accumulated sum related to the first set of qbit weights is stored in register *D1*. The accumulated sum related to the second set of qbit weights is stored in register *D2*. *D1* and *D2* are added providing the output of the quantum neuron to the *Shift* unit. Once evaluated by the *Shift* unit, the result is stored in register *D3*. In the mean time, the remaining half of the inputs is evaluated following the same procedure and the result is stored in register *D4*. Finally, *D4* and *D3* are added and the result is the output of the DN.

4.3.12 3-XOR Units Model Hardware Implementation

For this case, the implementation of the 3-XOR Units Model design is depicted in Figure 4.14 below:

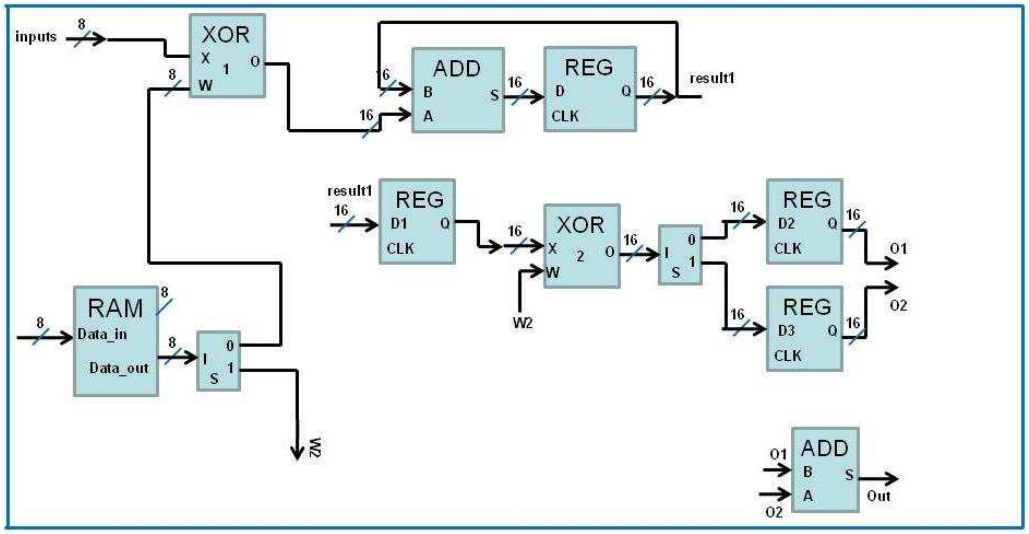


Figure 4.14 3-XOR Units Model Hardware Implementation

For this design, the first half of the inputs is evaluated by *XOR unit 1*, and the accumulated sum is stored in register *D1*. Then *D1* is sent to *XOR unit 2* to be evaluated and the result is stored in register *D2*. In the mean time, the second half of the inputs is evaluated following the same procedure and the result is stored in register *D3*. Finally, *D2* and *D3* are added and the result is the output of the DN.

4.3.13 3-Shift Units Model Hardware Implementation

The implementation of this model is very similar to the previous design and is shown in Figure 4.15. For this case, the first half of the inputs is evaluated by *Shift unit 1*, and the accumulated sum is stored in register *D1*. Then *D1* is sent to *Shift unit 2* to be evaluated and the result is stored in register D2. In the mean time, the second half of the inputs is evaluated following the same procedure and the result is stored in register *D3*. Finally, *D2* and *D3* are added and the result is the output of the DN.

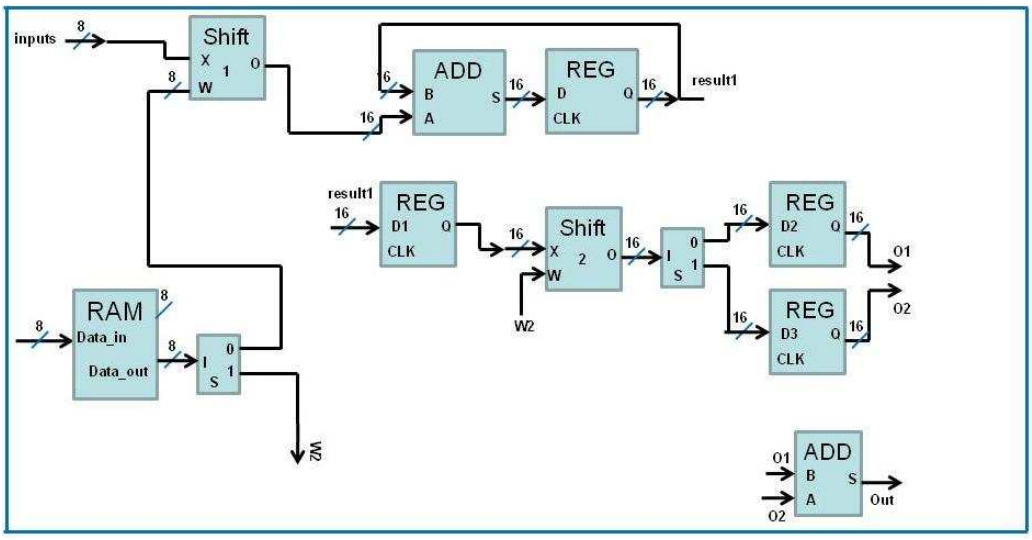


Figure 4.15 3-Shift Units Model Hardware Implementation

4.4 Hardware Implementation Results

Each of the DN models and the traditional perceptron models described in the previous section were implemented in VHDL using a Xilinx Spartan 3 FPGA as the reference hardware device. The hardware resources utilized by a single neuron model were recorded for the hidden layer and the output layer. Then based on the results described in chapter 3, the total resources utilized by a network considering a particular benchmark problem were calculated and recorded. Finally, based on these results, the models were ranked based on the amount of hardware resources utilized. Table 4.1 shows the performance results and the hardware implementation resources utilized by the different neuron models. *Hw-H* shows the resources utilized by a single neuron model in the hidden layer, and *Hw-O* shows the resources utilized by a traditional perceptron, since DNs are only used in the hidden layer. *Hw-T* shows the total resources utilized by the traditional network and the network using DNs calculated based on the wine benchmark problem. *Hw-R* shows the ranking of the total resources utilized by the different models implemented.

Table 4.1 Performance and Hardware Implementation Results

M	F(x)	Bench	H-Ns	Max	Avg	Min	P-R	Hw-H	Hw-O	Hw-T	Hw-R
Crisp	Bip-Sig	Wine	22	100%	61.86%	0.0%	10	3%	2%	70%	5
		Vowel		100%	27.02%	9.1%					
		Yeast		99.16%	61.08%	39.1%					
		Letter		61.63%	20.1%	3.9%					
Intial	Bip-Sig	Wine	5	100%	77.04%	0.0%	2	12%	2%	64%	3
		Vowel	8	100%	24.08%	9.1%					
		Yeast	8	100%	57.1%	39.1%					
		Letter	7	100%	15.13%	3.9%					
XOR	Bip-Sig	Wine	5	100%	81.28%	0.0%	6	12%	2%	64%	3
		Vowel	7	54.11%	15.53%	9.1%					
		Yeast	5	100%	61.13%	39.1%					
		Letter	7	100%	12.1%	3.9%					
Shift	Bip-Sig	Wine	5	100%	74.84%	0.0%	8	15%	2%	79%	7
		Vowel	11	98.92%	22.56%	9.1%					
		Yeast	5	99.16%	63.41%	39.1%					
		Letter	10	100%	14.6%	3.9%					
Quant	Bip-Sig	Wine	5	100%	92.99%	32.43%	3	13%	2%	69%	4
		Vowel	8	100%	17.33%	9.1%					
		Yeast	5	99.16%	61.94%	39.1%					
		Letter	7	100%	12.3%	3.9%					
XO-SH	Bip-Sig	Wine	5	100%	76.71%	0.0%	12	17%	2%	89%	9
		Vowel	7	72.73%	20.06%	9.1%					
		Yeast	7	99.16%	60.91%	39.1%					
		Letter	7	61.35%	11.84%	3.9%					
XO-QU	Bip-Sig	Wine	5	100%	84.15%	0.0%	7	16%	2%	84%	8
		Vowel	7	52.4%	21.1%	9.1%					
		Yeast	7	100%	61.92%	39.1%					
		Letter	8	100%	13.8%	3.9%					
SH-QU	Bip-Sig	Wine	5	100%	86.49%	0.0%	5	21%	2%	109%	10
		Vowel	8	72.73%	21.8%	9.1%					
		Yeast	8	100%	63.22%	39.1%					
		Letter	8	100%	13.55%	3.9%					
2X-QU	Bip-Sig	Wine	5	100%	35.23%	0.0%	4	16%	2%	84%	8
		Vowel	7	95.7%	41.87%	10.61%					
		Yeast	7	100%	55.28%	39.1%					
		Letter	7	100%	23.88%	7.28%					
2S-QU	Bip-Sig	Wine	8	100%	100%	100%	1	16%	2%	132%	11
		Vowel	7	100%	43.47%	10.61%					
		Yeast	7	100%	60.1%	39.1%					
		Letter	7	100%	23.88%	7.28%					
2Q-1X	Bip-Sig	Wine	5	100%	79.61%	0.0%	9	6%	2%	34%	2
		Vowel	7	72.73%	18.84%	9.1%					
		Yeast	8	99.16%	64.94%	39.1%					
		Letter	7	100%	17.54%	3.9%					
2Q-1S	Bip-Sig	Wine	8	100%	100%	100%	11	9%	2%	76%	6
		Vowel	7	100%	44%	9.1%					
		Yeast	7	89.1%	48.81%	39.1%					
		Letter	8	61.63%	18.19%	3.9%					
3-XOR	Bip-Sig	Wine	5	100%	81.01%	0.0%	5	4%	2%	24%	1
		Vowel	8	72.73%	23.43%	9.1%					
		Yeast	9	100%	44.15%	5.72%					
		Letter	9	100%	16.15%	3.9%					
3-SH	Bip-Sig	Wine	5	40.54%	40.54%	40.54%	13	13%	2%	69%	4
		Vowel	7	54.3%	23.97%	13.42%					
		Yeast	8	89.1%	83.6%	68.69%					
		Letter	7	54%	54%	54%					

The total amount of hardware resources utilized is calculated as shown in equation 4.1 below:

$$T = (n_h * r_h) + (n_o * r_o) \quad (4.1)$$

Where n_h is the number of neurons in the hidden layer, r_h is the percentage of the resources required by one neuron in the hidden layer, n_o is the number of neurons in the output layer, r_o is the percentage of the resources required by one output neuron, and T is the total percentage of resources utilized by the ANN to be implemented.

As indicated in table 4.1, based on the wine benchmark problem, there are 6 models that require less hardware resources than the traditional network. These models are the *initial DN model*, the *XOR-units model*, *the quantum neurons model*, *the two quantum neurons & one XOR unit model*, *the three shift-units model*, and *the three XOR-units model*, which is the design that required the least amount of hardware resources. The results found show that twelve models, including the traditional network, were able to fit on the FPGA selected. Only two models did not fit on the device selected, which only means that additional hardware resources were required by those models. Based on the results obtained, the DN model approach shows its potential to reduce the hardware resources required when compared to the traditional network, which is always a good advantage, considering how critical space is in hardware implementations, due to strict restrictions on FPGA capacity.

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The main motivation of this study has been to develop a neuron model in an attempt to increase the performance of a traditional feed forward MLP. Consequently, this motivation gave rise to the concept of the DN model, which has been introduced in this research study. Several DN models were developed, implemented, and tested using four different benchmark problems.

Based on the experiment results, it was concluded that in general, the use of the DNs in the output layer does not provide additional benefits in performance. Therefore, the DNs were only used in the hidden layer. In addition, several activation functions were implemented to observe their contribution to the performance of the networks. Based on the results found, the binary sigmoid function was selected. However, this does not mean that the binary sigmoid function will always provide the best results. The use of activation functions is application specific, and it is always a good idea to investigate the effects of different activation functions in order to select the one that provides the best results with regard to the specific application in question.

Even though there was not much room for improvement based on the results of the traditional network, ten out of the thirteen DN models successfully achieved better performance when compared to the traditional network. This shows that the DN model approach has the capability to increase the performance of a traditional feed forward MLP. Moreover, it is important to note that the use of quantum neurons inside some of the DN models, contributed greatly in increasing the performance of the traditional network.

Two additional aspects were considered to be observed in this research study. The first is the simulation time of the different DN models implemented when compared to the traditional network. Based on the results obtained, all DN models showed a smaller simulation time than the traditional network. This indicates that the different DN models developed provided less computational load, which is an additional asset provided by the DNs with regard to the four benchmark problems tested. The second aspect considered in this research study is the hardware implementation of the different DN models using a Xilinx Spartan 3 as the reference device. The results found with respect to the wine benchmark problem indicated that six out of the thirteen DN models utilized less hardware resources than the traditional neuron model. This indicates that potentially, the DN model approach may require less hardware resources than the traditional network, however, this is application specific.

Considering that the main objective of this research study was focused on increasing the performance of the traditional network, the experiment results indicate that in general, the DN approach achieved the goal successfully with regard to the four benchmark problems tested.

5.2 Future Work

Based on the DN models presented, suggested future work would be first of all to use the DNs in additional different applications to observe their performance. Moreover, future work for the DNs would be to use them in different neural network architectures to observe if the models presented perform as well as shown in this research work. Secondly, additional activation functions may be considered in an attempt to improve processing power of the DN models. Furthermore, as the experiment results indicate, future work dedicated to quantum neurons is highly recommended in order to increase the processing power of the DN models as well. In

addition, other learning algorithms may be examined to improve the convergence of the networks such as resilient propagation, potentially resulting in a reduction of the simulation time.

Also, with regard to the hardware implementation of the different DN models, future work may be focused on the design of more efficient and compact hardware in order to reduce the hardware resources utilized.

Finally, a future path to continue this research is to combine DNs with fuzzy logic using interval mathematics. It would be greatly interesting to observe if the resulting hybrid network is able to further increase the performance of the different networks tested in this research study.

BIBLIOGRAPHY

[ABD99] Abdi, Herve, Dominique Valentin, and Betty Edelman. "*Neural Networks: Quantitative Applications in the Social Sciences*". Thousand Oaks, Calif.: Sage Publications, 1999.

[AHM04] Ahmad, A.M., et al. "*Recurrent Neural Network with Backpropagation through Time for Speech Recognition*". IEEE International Symposium on Communications and Information Technology, vol. 1, pp. 98-102, October 2004.

[ALK08] Al-Kazzaz, Sa'ad Ahmed, and Rafid Ahmed Khalil. "*FPGA Implementation of Artificial Neurons: Comparison Study*". Third International Conference on Information and Communication Technologies: From Theory to Applications, pp. 1-6, April 2008.

[ALL09] Allen, J.N, H.S. Abdel-Aty-Zohdy, and R.L. Ewing. "*Cognitive Processing Using Spiking Neural Networks*". Proceedings of the IEEE National Aerospace & Electronics Conference, pp. 56-64, July 2009.

[ALV00] Alvarez Grima, Mario. "*Neuro-Fuzzy Modeling in Engineering Geology: Applications to Mechanical Rock Excavation, Rock Strength Estimation, and Geological Mapping*". Rotterdam, Netherlands: A.A.Balkema, 2000.

[AMB05] Ambaryan, Tigran R. "*Neuro-Quantum Networks in Tasks of Pattern Recognition*". Proceedings of the IEEE International Conference on Physics and Control, pp. 527-530, August 2005.

[BAI08] Bailey, Julian A., et al. "*Behavioral Simulation and Synthesis of Biological Neuron Systems using VHDL*". IEEE International Behavioral Modeling and Simulation Workshop, pp. 7-12, September 2008.

[BLU92] Blum, Adam. "*Neural Networks in C++: An Object-Oriented Framework for Building Connectionist Systems*". New York: Wiley, 1992.

[BOS96] Bose, N. K., and P. Liang. "*Neural Network Fundamentals with Graphs, Algorithms, and Applications*". New York: McGraw-Hill, 1996.

[BOU11] Bouguilla, N. "*Count Data Modeling and Classification using Finite Mixtures of Distributors*". IEEE Transactions on Neural Networks, vol. 22, pp. 186-198, February 2011.

[BRO01] Brown, B.D., and H.C. Card. "*Stochastic Neural Computation 1: Computational Elements*". IEEE Transactions on Computers, vol. 50, pp. 891-905, September 2001.

[BRO09] Brown, Stephen, and Zvonko Vranesic. "*Fundamentals of Digital Logic with*

VHDL Design". New York: McGraw-Hill, 2009.

[BUB11] Bubtiena, A.M., et al. "*Application of Artificial Neural Networks in Modeling Water Networks*". IEEE 7th International Colloquium on Signal Processing and Its Applications, pp. 50-57, March 2011.

[CAL04] Calles, Erubey. "*A Survey of Transfer Functions for Independent Speaker Vowel Recognition using Neural Networks*". Master's Thesis, The University of Texas at El Paso, 2004.

[CAO09] Cao, Maojun, and Fuhua Shang. "*Quantum Neural Networks with Application in Adjusting PID Parameters*". International Conference on Information Engineering and Computer Science, pp. 1-5, December 2009.

[CAP11] Capizzi, G., et al. "*Recurrent Neural Network-Based Modeling and Simulation of Lead-Acid Batteries Charge-Discharge*". IEEE Transactions on Energy Conversion, vol. 26, pp. 435-443, June 2011.

[CHA05] Chandana, S. and R.V. Mayorga. "*The New Rough Neuron*". International Conference on Neural Networks and Brain, vol. 1, pp. 13-18, October 2005.

[CHA11] Chaofeng, Li, et al. "*Blind Image Quality Assessment using a General Regression Neural Network*". IEEE Transactions on Neural Networks, vol. 22, pp. 793-799, May 2011.

[CHE07] Chen, Goong, et al. "*Quantum Computing Devices: Principles, Design, and Analysis*". Boca Raton, FL: Chapman & Hall/CRC, 2007.

[CHO07] Chow, Tommy W.S., and Siu-Yeung Cho. "*Neural Networks and Computing: Learning Algorithms and Applications*". London: Imperial College Press, 2007.

[CHO10] Choudhary, A., et al. "*Performance Analysis of Feed Forward MLP with Various Activation Functions for Handwritten Numerals Recognition*". 2nd International Conference on Computer and Automation Engineering, vol. 5, pp. 852-856, February 2010.

[CHU08] Chu, Pong P. "*FPGA Prototyping by VHDL examples: Xilinx Spartan-3 version*". Hoboken, N.J.: Wiley-Interscience, 2008.

[DAW09] Dawwd, S.A., and B.S. Mahmood. "*A Reconfigurable Interconnected Filter for Face Recognition Based on Convolution Neural Network*". 4th International Design and Test Workshop, pp. 1-6, November 2009.

[DEL99] Del Carmen Lucero, Ivonne. "*A Systematic Method for Automatic Generation of Membership Functions for Fuzzy Logic Applications*". Master's Thesis, The University of Texas at El Paso, 1999.

[DEL10] Delshad, E., et al. "*Spiking Neural Network Learning Algorithms: Using Learning Rates Adaptation of Gradient and Momentum Steps*". 5th International Symposium on Telecommunications, pp. 944-949, December 2010.

[DEW03] Dewhurst, Stephen C. "*C++ Gotchas: Avoiding Common Problems in Coding and Design*". Boston: Addision-Wesley, 2003.

[DON10] Dong, Yan, et al. "*System Identification and Analysis of the Frequency Characteristic of the Airborne Stabilizing Platform*". 2nd International Conference on Computer Engineering and Technology, vol. 7, pp. 409-413, April 2010.

[EFE09] Efe, M.O. "*A Type 2 Neuron Model for Classification and Regression Problems*". Proceedings of the 4th International IEEE EMBS Conference on Neural Engineering, pp. 677-680, April, 2009.

[EIC06] Eickhoff, R., et al. "*SIRENS: A Simple Reconfigurable Neural Hardware Structure for Artificial Neural Network Implementations*". International Joint Conference on Neural Networks, pp. 2830-2837, October 2006.

[FAL07] Fallahpour, M., et al. "*A Supervisory Control Fuzzy Control of Back-End Temperature of Rotary Cement Kilns*". International Conference on Control, Automation and Systems, pp. 429-434, October, 2007.

[FAU94] Fausett, Laurene V. "*Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*". Upper Saddle River, NJ: Prentice Hall, 1994.

[FAY10] Fayad, Ramzi. "*Tool Monitoring for Drilling Process Applying Enhanced Neural Networks*". The 2nd International Conference on Computer and Automation Engineering, vol. 3, pp. 200-204, February 2010.

[GER08] Geretti, Luca, and Antonio Abramo. "*The Correspondence between Deterministic and Stochastic Digital Neurons: Analysis and Methodology*". IEEE Transactions on Neural Networks, vol. 19, pp. 1739-1752, October 2008.

[GOK05] Gokhale, Maya B., and Paul S. Graham. "*Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*". Dordrecht: Springer, 2005.

[GRA09] Gradojevic, Nikola, Ramazan Gencay, and Dragan Kukolj. "*Option Pricing With Modular Neural Networks*". IEEE Transactions on Neural Networks, vol. 20, pp. 626-637, April 2009.

[GUP09] Gupta, V., et al. "*FPGA Design and Implementation Issues of Artificial Neural Network Based PID Controllers*". International Conference on Advances in Recent Technologies in Communication and Computing, pp. 860-862, October 2009.

[GUP11] Gupta, Niloy, et al. "*Fuzzy File Management*". 3rd International Conference on Electronics Computer Technology, vol. 1, pp. 225-228, April 2011.

[HAS09] Hashimoto, Sho, and Hiroyuki Torikai. "*Bifurcation Analysis of a Reconfigurable Hybrid Spiking Neuron and its Novel Online Learning Algorithm*". International Joint Conference on Neural Networks, pp. 1134-1141, June 2009.

[HAU08] Hauck, Scott, and Andre De Hon, eds. "*Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*". Boston: Morgan Kaufmann, 2008.

[HEL08] Hell, Michel, Pyramo Costa Jr., and Fernando Gomide. "*Hybrid Neurofuzzy Computing with Nullneurons*". IEEE International Joint Conference on Neural Networks, pp. 3653-3659, June 2008.

[HIG06] Higuchi, Tetsuya, Yong Liu, and Xin Yao, eds. "*Evolvable Hardware*". New York: Springer, 2006.

[HIK03] Hikawa, Hiroomi. "*A New Digital Pulse-Mode Neuron with Adjustable Activation Function*". IEEE Transactions on Neural Networks, vol. 14, pp. 236-242, January 2003.

[HIS11] Hishiki, T., and H. Torikai. "*A Novel Rotate-and-Fire Digital Spiking Neuron and its Neuron-Like Bifurcations and Responses*". IEEE Transactions on Neural Networks, vol. 22, pp. 752-767, May 2011.

[HOH04] Hohmann, S.G., et al. "*Training Fast Mixed-Signal Neural Networks for Data Classification*". IEEE International Joint Conference on Neural Networks, vol. 4, pp. 2647-2652, July 2004.

[HOM02] Homma, N., and M.M. Gupta. "*Study on General Second-Order Neural Units (SONUs)*". Proceedings of the 5th Biannual World Automation Congress, vol. 13, pp. 177-182, 2002.

[HU10] Hu, Yinquan, et al. "*Design of BPNN Speed PI Controller based on GA*". IEEE International Conference on Automation and Logistics, pp. 1474-1478, August 2009.

[HUN11] Hunter, David, and Bogdan Wilamowski. "*Parallel Multi-Layer Neural Network Architecture with Improved Efficiency*". 4th International Conference on Human System Interactions, pp. 299-304, May 2011.

[INO05] Inoue, H., and H. Narihisa. "*Incremental Learning Using Self-Organizing Neural Grove*". IEEE-EURASIP Nonlinear Signal and Image Processing, May 2005.

[ISL08] Islam, M.M., et al. "*Bagging and Boosting Negatively Correlated Neural Networks*". IEEE Transactions on Systems, Man, and Cybernetics, vol. 38, pp.

771-784, June 2008.

[JAN93] Jang, Jyh-Shing Roger. "*ANFIS: Adaptive-Network-Based Fuzzy Inference System*". IEEE Transactions on Systems, Man, and Cybernetics, vol. 23, pp. 665-685, May 1993.

[JEN01] Jeney, G., J. Levendovszky, and L. Kovacs. "*Blind Adaptive Stochastic Neural Network for Multiuser Detection*". 53rd IEEE Vehicular Technology Conference, vol. 3, pp. 1868-1872, May 2001.

[JIA07] Jiang, Wei, and Seong G. Kong. "*Block-Based Neural Networks for Personalized ECG Signal Classification*". IEEE Transactions on Neural Networks, vol. 18, pp. 1750-1761, November 2007.

[JIE09] Jie, Tian, et al. "*Classification of Underwater Objects Base on Probabilistic Neural Network*". Fifth International Conference on Natural Computation, vol. 2, pp. 38-42, August 2009.

[JOY07] Joy Vasantha Rani, S.P., and P. Kanagasabapathy. "*Multilayer Perceptron Neural Network Architecture using VHDL with Combinational Logic Sigmoid Function*". International Conference on Signal Processing, Communications, and Networking, pp. 404-409, February 2007.

[KAN11] Kang, Peng, and Zhao Jin. "*Speed Control of Induction Motor Using Neural Network Sliding Mode Controller*". International Conference on Electric Information and Control Engineering, pp. 6125-6129, April 2011.

[KAR96] Kartalopoulos, Stamatios V. "*Understanding Neural Networks and Fuzzy Logic: Basic Concepts and Applications*". New York: IEEE Press, 1996.

[KAW00] Kawaguchi, Kiyoshi. "*A Multithreaded Software Model for Backpropagation Neural Network Applications*". Master's Thesis, The University of Texas at El Paso, 2000.

[KAY07] Kaye, Phillip, Raymond Laflamme, and Michele Mosca. "*An Introduction to Quantum Computing*". New York: Oxford University Press, 2007.

[KIM10] Kim, HyungTae, et al. "*Quick Wafer Alignment Using Feedforward Neural Networks*". IEEE Transactions on Automation Science and Engineering, vol. 7, pp. 377-382, April 2010.

[KLI95] Klir, George J. and Yuan Bo. "*Fuzzy Sets and Fuzzy Logic: Theory and Applications*". Upper Saddle River, NJ: Prentice Hall, 1995.

[KON04] Kong, Seong G. "*Time Series Prediction with Evolvable Block-based Neural Networks*". Proceedings of the IEEE International Joint Conference on Neural

Networks, vol. 2, pp. 1579-1583, July 2004.

[KOS97] Kosko, Bart. "*Fuzzy Engineering*". Upper Saddle River, NJ: Prentice Hall, 1997.

[KOT08] Kothari, Ashwin, et al. "*Rough Neuron Based Neural Classifier*". First International Conference on Emerging Trends in Engineering and Technology, pp. 624-628, July 2008.

[KRA07] Kraipeerapun, P., et al. "*Quantification of Vagueness in Multiclass Classification Based on Multiple Binary Neural Networks*". International Conference on Machine Learning and Cybernetics, vol. 1, pp. 140-144, August 2007.

[KUM99] Kumar, S., et al. "*A Versatile Framework for Labelling Imagery with a Large Number of Classes*". International Joint Conference on Neural Network, vol. 4, pp. 2829-2833, July 1999.

[LEE04] Lee, Chang-Der, et al. "*The Non-Stationary Signal Prediction by Using Quantum NN*". IEEE International Conference on Systems, Man and Cybernetics, vol. 4, pp. 3291-3295, October 2004.

[LEI08] Leiner, Barba J., et al. "*Hardware Architecture for FPGA Implementation of a Neural Network and its Application in Images Processing*". IEEE Conference on Electronics, Robotics and Automotive Mechanics, pp. 405-410, October 2003.

[LEI09] Leite, D.F, et al. "*Evolving Granular Classification Neural Networks*". International Joint Conference on Neural Networks, pp. 1736-1743, June 2009.

[LEI11] Leila, Chergui, et al. "*Combining Neural Networks for Arabic Handwriting Recognition*". 10th International Symposium on Programming and Systems, pp. 74-79, April 2011.

[LI02] Li, Fei, Shengmei Zhao, and Baoyu Zheng. "*Quantum Neural Network in Speech Recognition*". Proceedings of the International Conference on Signal Processing, vol. 2, pp. 1267-1270, August 2002.

[LI03] Li, Fei, and Baoyu Zheng. "*A Study of Quantum Neural Networks*". IEEE International Conference on Neural Networks & Signal Processing, Nanjing, China, vol. 1, pp. 539-542, December 2003.

[LI04] Li, Fei, et al. "*A Learning Algorithm for Quantum Neuron*". Proceedings of the International Conference on Signal Processing, vol. 2, pp. 1538-1541, September 2004.

[LI05] Li, Fei, Shengmei Zhao, and Baoyu Zheng. "*Feedback Quantum Neuron and Its Application*". Proceedings of the IEEE International Conference on Neural Networks and Brain, vol. 2, pp. 867-871, October 2005.

[LI09] Li, Fei, and Guobiao Xu. "*A Novel Scheme of Speech Enhancement Based on Quantum Neural Network*". International Asia Symposium on Intelligent Interaction and Affective Computing, pp. 141-144, December 2009.

[LI11] Li, Li, and Feng Guangli. "*The License Plate Recognition System Based on Fuzzy Theory and BP Neural Network*". International Conference on Intelligent Computation Technology and Automation, vol. 1, pp. 267-271, March 2011.

[LIK11] Li, Keyang, et al. "*Based on Hopfield Neural Network to Determine The Air Quality Levels*". International Conference on Business Management and Electronic Information, vol. 4, pp. 182-185, May 2011.

[LIL10] Li, Lei, et al. "*Natural Gradient Algorithm Based on a Class of Activation Functions and its Applications in BSS*". Proceedings of the Fifth International Conference on Machine Learning and Cybernetics, pp. 2985-2989, August 2006.

[LIN04] Lin, Cheng-Jian, Cheng-Hung Chen, and Chi-Yung Lee. "*A Self-Adaptive Quantum Radial Basis Function Network for Classification Applications*". Proceedings of the IEEE International Joint Conference on Neural Networks, vol.4, pp. 3263-3268, July 2004.

[LIU08] Liu, Xian-de, and Huan Li. "*A Weighted Fuzzy Reasoning Process Neural Network and its Application*". IEEE Fourth International Conference on Natural Computation, vol. 3, pp. 59-63, October 2008.

[LIU09] Liu, Jindong, et al. "*A Biometric Spiking Neural Network of the Auditory Midbrain for Mobile Robot Sound Localization in Reverberant Environments*". International Joint Conference on Neural Networks, pp. 1855-1862, June 2009.

[LIU10] Liu, Lijuan, and Mingrong Deng. "*An Evolutionary Artificial Neural Network Approach for Breast Cancer Diagnosis*". Third International Conference on Knowledge Discovery and Data Mining, pp. 593-596, 2010.

[LIU11] Liu, Qingshan, and Jun Wang. "*Finite-Time Convergent Recurrent Neural Network with a Hard-Limiting Activation Function for Constrained Optimization with Piecewise-Linear Objective Functions*". IEEE Transactions on Neural Networks, vol. 22, pp. 601-613, April 2011.

[LIY08] Li, Yanlai, Kuanquan Wang, and Tao Li. "*Modular Neural Network Structure with Fast Training/Recognition Algorithm for Pattern Recognition*". IEEE International Conference on Granular Computing, pp. 401-406, August 2008.

[LOP08] Lopez, M., and P. Melin. "*Topology Optimization of Fuzzy Systems for Response Integration in Ensemble Neural Networks: The Case of Finger Print Recognition*". Annual Meeting of the North American Fuzzy Information Processing Society, pp.1-6, 2008.

[LOR08] Lorrentz, P., et al. "*An FPGA Based Adaptive Weightless Neural Network Hardware*". NASA/ESA Conference on Adaptive Hardware and Systems, pp. 220-227, June 2008.

[LUD10] Ludwig, O., and Urbano Nunes. "*Novel Maximum-Margin Training Algorithms for Supervised Neural Networks*". IEEE Transactions on Neural Networks, vol. 21, pp. 972-984, June 2010.

[MAD07] Maduko, Elizabeth. "*Development and Testing of a Neuro-Fuzzy Classification System for IOS Data in Asthmatic Children*". Master's Thesis, The University of Texas at El Paso, 2007.

[MAN01] Mandic, Danilo P., and Jonathon A. Chambers. "*Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures, and Stability*". New York: John Wiley, 2001.

[MAR08] Marzi, Hosein. "*Modular Neural Network Architecture for Precise Condition Monitoring*". IEEE Transactions on Instrumentation and Measurement, vol. 57, pp. 805-812, April 2008.

[MAS93] Masters, Timothy. "*Practical Neural Networks Recipies in C++*". Boston: Academic Press, 1993.

[MCM08] McMahon, David. "*Quantum Computing Explained*". Hoboken, N.J.: Wiley-Interscience, 2008.

[MEN10] Meng, Ke, et al. "*A Self-Adaptive RBF Neural Network Classifier for Transformer Fault Analysis*". IEEE Transactions on Power Systems, vol. 25, pp. 1350-1360, August 2010.

[MIN02] Ming, Zhang, et al. "*Neuron-Adaptive Higher Order Neural-Network Models for Automated Financial Data Modeling*". IEEE Transactions on Neural Networks, vol. 13, pp. 188-204, January 2002.

[MIS08] Mishra, S., and S.K, Patra. "*Short Term Load Forecasting using a Novel Recurrent Neural Network*". IEEE Region 10 Conference TENCON 2008, pp. 1-6, November 2008.

[MIT02] Mitrpanont, Jarernsri L., and Ananta Srisuphab. "*The Realization of Quantum Complex-Valued Backpropagation Neural Network in Pattern Recognition Problem*". Proceedings of the 9th International Conference on Neural Information Processing, vol. 1, pp. 462-466, November 2002.

[MKA11] Mkadem, F., and S. Boumaiza. "*Physically Inspired Neural Network Model for RF Power Amplifier Behavioral Modeling and Digital Predistortion*". IEEE Transactions on Microwave Theory and Techniques, vol. 59, pp. 913-923,

January 2011.

[MOR06] Mori, Katsuhiro, et al. "*Qubit Inspired Neural Network towards Its Practical Applications*". International Joint Conference on Neural Networks, pp. 224-229, July 2006.

[MOR08] Morsch, Oliver. "*Quantum Bits and Quantum Secrets: How Quantum Physics is Revolutionizing Codes and Computers*". Weinheim, Germany: Wiley-VCH, 2008.

[MOR09] Moreno, Felix, et al. "*Reconfigurable Hardware of a Shape Recognition System Based on Specialized Tiny Neural Networks with On-Line Training*". IEEE Transactions on Industrial Electronics, vol. 56, pp. 2353-3263, August, 2009.

[MUN06] Munoz, D.M., et al. "*Implementation, Simulation, and Validation of Dispatching Algorithms for Elevator Systems*". IEEE International Conference on Reconfigurable Computing and FPGA's, pp. 1-8, September 2006.

[NAR07] Narain, Seema, and Ashu Jain. "*Artificial Neuron Models for Hydrological Modeling*". Proceedings of International Joint Conference on Neural Networks, pp. 1338-1342, August 2007.

[NAV96] Nava, Patricia A., and Javin M. Taylor. "*Speaker Independent Voice Recognition with a Fuzzy Neural Network*". Proceedings of the Fifth IEEE International Conference on Fuzzy Systems, vol. 3, pp. 2049-2052, September 1996.

[NAV98] Nava, Patricia A. "*Implementation of Neuro-Fuzzy Systems through Interval Mathematics*". Proceedings of the IEEE Joint Conference on Computational Intelligence in Robotics and Automation, Intelligent Systems and Semiotics, and Intelligent Control, pp. 365-369, September 1998.

[NED05] Nedjah, Nadia, and Luiza de Macedo Mourelle, eds. "*Evolvable Machines: Theory & Practice*". Berlin: Springer, 2005.

[NEL90] Nelson, Marilyn McCord and Illingworth, W.T. "*A Practical Guide to Neural Nets*". Reading, MA: Addison-Wesley Publishing Company, 1990.

[NIE11] Nie, Xiaobing, and Jinde Cao. "*Coexistence and Local Stability of Multiple Equilibria in Competitive Neural Networks with Nondecreasing Activation Functions*". International Conference on Information Science and Technology, pp. 70-76, March 2011.

[NOO03] Noory, Babak, and Voicu Groza. "*A Reconfigurable Approach Hardware Implementation of Neural Networks*". IEEE Canadian Conference on Electrical and Computer Engineering, vol. 3, pp. 1861-1864, May 2003.

[ONO09] Onomi, T., et al. "*Implementation of High-Speed Single Flux-Quantum Up/Down Counter for the Neural Computation Using Stochastic Logic*". IEEE Transactions on Applied Superconductivity, vol. 19, pp. 626-629, June 2009.

[ORL11] Orlowska-Kowalska, T., and M. Kaminski. "*FPGA Implementation of the Multilayer Neural Network for the Speed Estimation of the Two-Mass Drive Systems*". IEEE Transactions on Industrial Informatics, pp. 1-10, July 2011.

[PAN11] Panagiotakopoulos, C., and P. Tsampouka. "*The Margitron: A Generalized Perceptron with Margin*". IEEE Transactions on Neural Networks, vol. 22, March 2011.

[PAP11] Papageorgiou, E. I., "*Learning Algorithms for Fuzzy Cognitive Maps-A Review Study*". IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, pp. 1-14, May 2011.

[PED06] Pedrycz, Witold, Marek Reformat, and Kuwen Li. "*OR/AND Neurons and the Development of Interpretable Logic Models*". IEEE Transactions on Neural Networks, vol. 17, pp. 636-658, May 2006.

[PER08] Perez-Gallardo, Jorge R., et al. "*Interpretation of Mammographic Using Fuzzy Logic for Early Diagnosis of Breast Cancer*". Seventh Mexican International Conference on Artificial Intelligence, pp. 278-283, October 2008.

[PIT09] Pitti, Alexandre, et al. "*Contingency Perception and Agency Measure in Visuo-Motor Spiking Neural Networks*". IEEE Transactions on Autonomous Mental Development, vol. 1, pp. 86-97, May 2009.

[POS09] Postolache, Octavian A., J.M. Dias Pereira, and P.M.B. Silva Girao. "*Smart Sensors Network for Air Quality Monitoring Applications*". IEEE Transactions on Instrumentation and Measurement, vol. 58, pp. 3253-3262, September 2009.

[QIN11] Qing, Sun, et al. "*Implementation Study of an Analog Spiking Neural Network for Assisting Cardiac Delay Prediction in a Cardiac Resynchronization Therapy Device*". IEEE Transactions on Neural Networks, vol. 22, pp. 858-869, June 2011.

[RAG09] Raguraman, S.M., D. Tamilselvi, and N. Shivakumar. "*Mobile Robot Navigation Using Fuzzy Logic Controller*". International Conference on Control, Automation, Communication, and Energy Conservation, pp. 1-5, June 2009.

[RAJ10] Rajini, G.K., and G.R. Reddy. "*Performance Evaluation of Neural Networks for Shape Identification in Image Processing*". International Conference on Signal Acquisition and Processing, pp. 255-258, February 2010.

[RAV07] Ravi, V., et al. "*On-Line Evolving Fuzzy Clustering*". International Conference on

Computational Intelligence and Multimedia Applications, vol. 1, pp. 347-351, December 2007.

[RIC09] Rice, K.L., et al. "*FPGA Implementation of Izhikevich Spiking Neural Networks for Character Recognition*". International Conference on Reconfigurable Computing and FPGAs, pp. 451-456, December 2009.

[ROG97] Rogers, Joey. "*Object-Oriented Neural Networks in C++*". San Diego, CA: Academic Press, 1997.

[ROJ96] Rojas, Raul. "*Neural Networks: A Systematic Introduction*". New York: Springer-Verlag, 1996.

[SAA11] Saad, Z., et al. "*Modeling and Forecasting of Injected Fuel Flow Using Neural Network*". IEEE 7th International Colloquium on Signal Processing and Its Applications, pp. 243-247, March 2011.

[SAE03] Saenz, Jovan. "*Independent Speaker Vowel Recognition Using an Interval Neural Network and Its Hardware Implementation*". Master's Thesis, The University of Texas at El Paso, 2007.

[SAM07] Samarasinghe, Sandhya. "*Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition*". Boca Raton, FL: Auerbach, 2007.

[SHA11] Shamsuddin, N., and M.N. Taib. "*Diagnosis of Heart Diseases using Nonlinear ARX Model*". IEEE 7th International Colloquium on Signal Processing and Its Applications, pp. 388-394, March 2011.

[SHI10] Shiblee, Mohd, et al. "*Generalized Power Mean Neuron Model*". Third International Conference on Knowledge Discovery and Data Mining, pp. 276-279, January 2010.

[SAN02] Sanchez, E.N., and J.P. Perez. "*Stabilization of Stochastic Recurrent Neural Networks*". Proceedings of the 2002 IEEE International Symposium on Intelligent Control, pp. 445-447, October 2002.

[SAS09] Sasaki, H., and N. Kubota. "*Distributed Behavior Learning of Multiple Mobile Robots Based on Spiking Neural Network and Steady-State Generic Algorithm*". IEEE Workshop on Robotic Intelligence in Informationally Structured Space, pp. 73-78, March 2009.

[SEO03] Seow, Ming-Jung, Hau Ngo, and Vijayan Asari. "*Systolic Array Implementation of Block Based Hopfield Neural Network for Pattern Association*". Proceedings of the IEEE Computer Society Annual Symposium on VLSI, pp. 213-214, February 2003.

[SHI97] Shin, Y., et al. "*A Complex Pi-Sigma Network and Its Application to Equalization of Nonlinear Satellite Channels*". International Conference on Neural Networks, vol. 1, pp. 148-152, June 1997.

[SHI09] Shi, Yu, et al. "*Chaotic Synchronization of Coupled Hindmarsh-Rose Neurons Using Adaptive Control*". International Conference on Biomedical Engineering and Informatics, pp. 1-5, October 2009.

[SIL08] Silva, P.H.F., and A.L.P.S. Campos. "*Fast and Accurate Modeling of Frequency-Selective Surfaces using a new Modular Neural Network Configuration of Multilayer Perceptrons*". IET Microwaves, Antennas & Propagation 2.5 (2008): 503-511.

[SIN01] Sinha, M., et al. "*A Compensatory Wavelet Neuron Model*". Joint 9th IFSA World Congress and 20th NAFIPS International Conference, vol. 3, pp. 1372-1377, July 2001.

[SON07] Song, Tao, et al. "*A Modified Probabilistic Neural Network for Partial Volume Segmentation in Brain MR Image*". IEEE Transactions on Neural Networks, vol. 18, pp. 1424-1432, September 2007.

[SOR11] So, Rosa Q., et al. "*Irregular High Frequency Patterns Decrease The Effectiveness of Deep Brain Stimulation in a Rat Model of Parkinson's Disease*". 5th International IEEE/EMBS Conference on Neural Engineering, pp. 322-325, April 2011.

[SPA04] Spasojevic, B. "*Fuzzy Optical Sensor for Washing Machine*". 7th Seminar on Neural Network Applications in Electrical Engineering, pp. 237-242, September, 2004.

[SPE92] Specht, D.F. "*Enhancements to Probabilistic Neural Networks*". International Joint Conference on Neural Networks, vol. 1, pp. 761-768, June 1992.

[STA07] Stavrakoudis, Dimitris G., and John B. Theocharis. "*Pipelined Recurrent Fuzzy Neural Networks for Nonlinear Adaptive Speech Prediction*". IEEE Transactions on Systems, Man, and Cybernetics, vol. 37, pp. 1305-1320, October 2007.

[TAT10] Tatsumi, k., et al. "*Nonlinear Extension of Multiobjective Multiclass Support Vector Machine*". IEEE International Conference of Systems, Man, and Cybernetics, pp. 1338-1343, October 2010.

[TIA09] Tian, Jie, Shanhua Xue, and Haining Huang. "*Classification of Underwater Objects Based on Probabilistic Neural Network*". Fifth International Conference on Natural Computation, vol.2, pp. 38-42, August 2009.

[TOR09] Torikai, Hiroyuki, and Toru Nishigami. "*A Novel Chaotic Spiking Neuron and its*

Paralleled Spike Encoding Function". International Joint Conference on Neural Networks, pp. 3132-3139, June 2009.

[TRI10] Tripathy, M., et al. "Power Transformer Differential Protection Based on Optimal Probabilistic Neural Network". IEEE Transactions on Power Delivery, vol. 25, pp. 102-112, January 2010.

[TRI11] Tripathi, B.K., and P.K. Kalra. "*On Efficient Learning Machine with Root-Power Mean Neuron in Complex Domain*". IEEE Transactions on Neural Networks, vol. 22, pp. 727-738, May 2011.

[TSA05] Tsai, Xin-Yi, et al. "*Quantum NN vs. NN in Signal Recognition*". Proceedings of the Third International Conference on Information Technology and Applications, vol. 1, pp. 308-312, July 2005.

[TUK10] Tukel, M., and M.E. Yalcin. "*A New Architecture for Cellular Neural Network on Reconfigurable Hardware with an Advance Memory Allocation Method*". 12th International Workshop on Cellular Nanoscale Networks and their Applications, pp. 1-6, February 2010.

[UCI10] UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, http://archive.ics.uci.edu/ml, 2010.

[UDR05] Udrescu, Mihai, Lucian Prodan, and Mircea Vladutiu. "*The Bubble Bit Technique as Improvement of HDL-Based Quantum Circuits Simulation*". Proceedings of the 38th Annual Simulation Symposium, pp. 217-224, April 2005.

[VIL11] Vilovic, I., and Burum, N. "*A Comparison of MLP and RBF Neural Networks Architectures for Electromagnetic Field Prediction in Indoor Environments*". Proceedings of the 5th European Conference on Antennas and Propagation, pp. 1719-1723, April 2011.

[VOL11] Volyanskyy, K.Y., et al. "*Pressure-and Work-Limited Neuroadaptive Control for Mechanical Ventilation of Critical Care Patients*". IEEE Transactions on Neural Networks, vol. 22, pp. 614-626, April 2011.

[WAN05] Wang, Kejun, and Mo Tang. "*A Study of Fuzzy Chaotic Neuron*". International Conference on Neural Networks and Brain, vol. 1, pp. 292-297, October 2005.

[WAN08] Wang, Jiang-Jiang, et al. "*Self-Adaptive Neuron PID Control in Exhaust Temperature of Micro Gas Turbine*". International Conference on Machine Learning and Cybernetics, vol. 4, pp. 2125-2130, July 2008.

[WAS93] Wasserman, Philip D. "*Advanced Methods in Neural Computing*". New York: Van Nostrand Reinhold, 1993.

[WUJ09] Wu, Jun, and Haiyan Shuai. "*Insulators ESDD Predicting Based on Wavelet Neural Network*". International Workshop on Intelligent Systems and Applications, pp. 1-4, 2009.

[WUQ09] Wu, QingXiang, et al. "*Detection of Straight Lines Using a Spiking Neural Network Model*". Fifth International Conference on Natural Computation, vol. 2, pp. 385-389, August 2009.

[XIA06] Xiang, Zhon-Ji, et al. "*Construction of Fuzzy Classification System Based on Multi-Objective Generic Algorithm*". 6th International Conference on Intelligent Systems Design and Applications, vol. 2, pp. 1029-1034, October 2006.

[XIA09] Xiao, Hong, and Maojun Cao. "*Hybrid Quantum Neural Networks Model Algorithm and Simulation*". 5th International Conference on Natural Computation, vol. 1, pp. 164-168, August 2009.

[XIA10] Xia, Jing-Hua, et al. "*Feedforward Neural Network Trained by BFGS Algorithm for Modeling Plasma Etching of Silicon Carbide*". IEEE Transactions on Plasma Science, vol. 38, pp. 142-148, February 2010.

[XIA11] Xianglei, Duan, et al. "*The Neural Network Direct Inverse Control of Four-Wheel Steering System*". 3rd International Conference on Measuring Technology and Mechatronics Automation, vol. 3, pp. 865-869, January 2011.

[XU07] Xu, Qing-Yang, et al. "*Gas Turbine Fault Diagnosis Based on Wavelet Neural Network*". Proceedings of the International Conference on Wavelet Analysis and Pattern Recognition, vol. 7, pp. 738-741, November 2007.

[XU08] Xu, Jie, and John G. Harris. "*The Time Derivative Neuron*". IEEE International Symposium on Circuits and Systems, pp. 436-439, May 2008.

[YAD03] Yadav, R.N., et al. "*Classification Using Single Neuron*". IEEE International Conference on Industrial Informatics, pp. 124-129, August 2003.

[YAD04] Yadav, R.N., et al. "*Multi-layer Neural Networks Using Generalized-Mean Neuron Model*". IEEE International Symposium on Communications and Information Technology, vol. 1, pp. 93-97, October 2004.

[YAL05] Yalamanchili, Sudhakar. "*VHDL: A Starter's Guide*". Upper Saddle River, N.J.: Prentice Hall, 2005.

[YAN11] Yanping, Fan, et al. "*Tank Unit Combat Formation Recognition Model Based on BP Neural Network in Virtual Simulation Training*". International Conference on Intelligent Computation Technology and Automation, vol. 1, pp. 952-955, March 2011.

[YEN99] Yen, John and Langari, Reza. "*Fuzzy Logic: Intelligence, Control, and Information*". New York: Prentice Hall, 1999.

[YIL11] Yilmaz, A., et al. "*Risk Analysis in Cancer Disease by Using Fuzzy Logic*". Annual Meeting of the North American Fuzzy Information Processing Society, pp. 1-5, March 2011.

[YOU08] Yousefian, N., et al. "*Speech Recognition with a Competitive Probabilistic Radial Basis Neural Network*". 4th International IEEE Conference on Intelligent Systems, vol. 1, pp. 7-19, September 2008.

[YOU09] Yu, Zhu, Zhang Hong, and Kong Ling-Dong. "*Research of Coal and Gas Outburst Forecasting Based on Immune Genetic Neural Network*". Second International Workshop on Knowledge Discovery and Data Mining, pp. 28-31, January 2009.

[ZAK97] Zaknich, A., and C.J.S. De Silva. "*Adaptive Learning Schemes of the Modified Probabilistic Neural Network*". 3rd International Conference on Algorithms and Architectures for Parallel Processing, pp. 597-610, December 1997.

[ZHA04] Zhao, Jie-Yu. "*Implementing Associate Memory with Quantum Neural Networks*". Proceedings of the Third Conference on Machine Learning and Cybernetics, vol. 5, pp. 3197-3200, August 2004.

[ZHA05] Zhang, Da, Hui Li, and Simon Y. Foo. "*A Simplified FPGA Implementation of Neural Network Algorithms Integrated with Stochastic Theory for Power Electronics Applications*". 31st Annual Conference of IEEE Industrial Electronics Society, pp. 1018-1023, November 2005.

[ZHA06] Zhao, Shengmei, JianHua Huang, and Baoyu Zheng. "*Recognition of Noisy English Letter by Quantum Back Propagation Network*". Proceedings of the 8th International Conference on Signal Processing, vol. 3, November 2006.

[ZUR92] Zurada, Jacek M. "Introduction to Artificial Neural Systems". St. Paul: West, 1992.

Appendix A

TRADITIONAL MLP SOURCE CODE

```
/*********************************************************
* Traditional Back-Propagation Feed Forward MLP          *
* File Name: non-int.c                                   *
*********************************************************/

#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"

/* ------- Constant Declaration: ----------------------- */
#define IN_UNTS    16                    /* no. of inputs */
#define HDN_UNTS   8       /* no. of hidden layer neurons */
#define OUT_UNTS   5             /* no. of output neurons */
#define FEED_IN    16    /* max no. of inputs to a neuron */
#define N_THRES    1.0           /* threshold for neurons */
#define MX_UNTS    16    /* max no. of neurons in any layer */
#define MX_PATTS   16000       /* max no. of input patterns */
#define MX_INPTS   16           /* max no. of input neurons */
#define MX_OUTPTS   5          /* max no. of output neurons */
#define NET_THRESHOLD 1.0
#define MAX 1.0
#define MIN 0.01

typedef struct {
                    float weight[FEED_IN +1];
                    float delta_weight[FEED_IN+1];
                    float delta;
                    float output;
                } neuron;

float input[MX_PATTS][MX_INPTS+1];
float goals[MX_PATTS][MX_OUTPTS];
float known_output[MX_PATTS][MX_OUTPTS];
float patt_err[MX_PATTS][MX_OUTPTS];
float tot_patt_err[MX_PATTS];
float sys_err;
float sumg;                                    /* for morlet, gaussian */
neuron hid_layer[HDN_UNTS+1];
neuron out_layer[OUT_UNTS];
neuron general_hid_layer[HDN_UNTS +1];         /* to hold initial weights */
neuron general_out_layer[OUT_UNTS];            /* to hold initial weights */
int num_samples, num_goals, num_misses = 0;
int max_iterations = 3000, min_err_iter;
int miss_flag = 0;
float learning_rate = 0.3, momentum = 0.3;
float lr = 0.0, mom = 0.0;
float max_tot_err = 0.01, max_indv_err = 0.001;
float min_err_td = 500.0;
FILE *rpt_ptr;
```

```
/*************************************************************
 * FUNCTION NAME: random_normalized()                        *
 *                                                           *
 * DESCRIPTION:  random number generator                     *
 *************************************************************/
float random_normalized(float minimum, float maximum, int flag)
{
      float ret_val;
      ret_val = (float) random()*(MAX-MIN)/(pow(2.0, 31.0)-1)+MIN;
      if (flag ==1)
            ret_val=ret_val*pow (-1.0,(double)random());
      return (ret_val);
}

/*************************************************************
* FUNCTION NAME: general_init_weights()                      *
*                                                            *
* DESCRIPTION:  initialize weights to ramdom values, then    *
*               store them and use them to provide constant  *
*               initial weights for all variations in        *
*               momentum and learning rate                   *
*************************************************************/
void general_init_weights()
{
  int j, k;
  float random_normalized(float, float, int);

  for(j=0; j< HDN_UNTS; j++){
     for(k=0; k< IN_UNTS +1; k++)
        general_hid_layer[j].weight[k] = random_normalized(MIN, MAX, 1);
  }                                        /* end loop of hidden units */
  for (j=0; j< OUT_UNTS; j++){
    for(k=0; k< HDN_UNTS +1; k++)
       general_out_layer[j].weight[k] = random_normalized(MIN, MAX, 1);
  }                                        /* end of output units */

}

/*************************************************************
 * FUNCTION NAME: init_weights()                             *
 *                                                           *
 * DESCRIPTION: initializes weights to random values.        *
 *************************************************************/
void init_weights()
{
   int j, k;

   /* set up hidden layer threshold neuron */
   hid_layer[HDN_UNTS].output = N_THRES;

      for(j=0; j < HDN_UNTS; j++){
        for(k=0; k< IN_UNTS+1; k++){
           hid_layer[j].weight[k] = general_hid_layer[j].weight[k];
        } /* endloop (# of weights for each unit) */
      }/* endloop (# of hidden units) */
      for(j=0; j < OUT_UNTS; j++){
        for(k=0; k< HDN_UNTS+1; k++){
```

```
            out_layer[j].weight[k] = general_out_layer[j].weight[k];
          }/* endloop (# weights for each unit) */
        }/* endloop (# of output units) */
}

/*************************************************************
 * FUNCTION NAME: init_inputs()                              *
 *                                                           *
 * DESCRIPTION: reads test patterns to be evaluated.         *
 *************************************************************/
void init_inputs()
{
    int j, k;
    FILE *f_ptr;


    if ((f_ptr = fopen("letter_bip_tst.txt", "r")) == NULL) {
      fprintf(rpt_ptr, "\nThe input test file could");
      fprintf(rpt_ptr, " not be opened. Terminating program.");
      exit(0);
    }/* endif */

    fscanf(f_ptr, "%d", &num_samples);
    for(j=0; j < num_samples; j++){
      for(k=0; k < IN_UNTS; k++){
              fscanf(f_ptr, "%f", &input[j][k]);

      }/* endloop (# of input units) */
      /* set up threshold "neuron" for input layer */
      input[j][IN_UNTS] = 1.0;
      /* ---------- this section was added for eval. purposes ---*/
      for(k=0; k < OUT_UNTS; k++){
         fscanf(f_ptr, "%f", &known_output[j][k]);
        }/* endloop (# of target output units) */
      /* ---------- end of section ------------------------------*/
    }/* endloop (# of data sets) */
    fclose(f_ptr);
}

/*************************************************************
 * FUNCTION NAME: propagate(i)                               *
 *                                                           *
 * DESCRIPTION:  propagates pattern #i through the network.  *
 *************************************************************/
void propagate(i)
  int i;
{
    int j, k;
    float sum, fnet;
    float tem;            /* for morlet wavelet */

    /* ---------- initialize & calc. hidden units' responses --- */
    for (j=0; j < HDN_UNTS; j++){
       sum = 0.0;
       for (k=0; k < IN_UNTS+1; k++){
           sum += hid_layer[j].weight[k] * input[i][k];
       }/* endloop (summing weighted inputs) */
```

```
       fnet = - (sum)/NET_THRESHOLD;
      /* hid_layer[j].output = 1.0/(1.0 + exp(fnet)); *//* binary sigmoid */
      /*hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;*/
/*bipolar sigmoid*/
      /*hid_layer[j].output=(exp(sum)-exp(fnet))/(exp(sum) + exp(fnet) );  */
/* hyperbolic tangent */
     /*if(hid_layer[j].output != hid_layer[j].output){ *//*if output is nan*/
            /* hid_layer[j].output = 1.0;  */
      /*  } */
      /*  hid_layer[j].output = exp(-(sum * sum)/2);*//* gaussian function */
      tem = ((sum)/3.0) * ((sum)/3.0);               /* for morlet wavelet */
      hid_layer[j].output=cos(1.75*((sum)/3.0))*exp(-tem/2);
/* morlet wavelet */
    }/* endloop (calculations of hidden layer) */

    /* ---------- initialize & calc. output units' responses --- */
    for (j=0; j < OUT_UNTS; j++){
      sum = 0.0;
            for (k=0; k < HDN_UNTS+1; k++){
        sum += out_layer[j].weight[k] * hid_layer[k].output;
      }/* endloop (summing weighted inputs) */

       fnet  = - (sum)/NET_THRESHOLD;
      /* out_layer[j].output = 1.0/(1.0 + exp(fnet));*/  /* binary sigmoid */
      /*out_layer[j].output=2.0/(1.0+exp(fnet))-1.0;*/  /* bipolar sigmoid*/
      /*out_layer[j].output=(exp(sum)-exp(fnet))/(exp(sum) + exp(fnet) ); */
/* hyperbolic tangent */
      /*if(out_layer[j].output != out_layer[j].output) { */
/* if output is nan */
         /*  out_layer[j].output = 1.0; */
      /*  } */
      /* out_layer[j].output = exp(-(sum * sum)/2); *//* gaussian function */
       tem = ((sum)/3.0) * ((sum)/3.0);              /* for morlet wavelet */
       out_layer[j].output = cos(1.75 * ((sum)/3.0)) * exp(-tem/2);
/* morlet wavelet */
       sumg = sum;                /* for gaussian/morlet wavelet function */
    }/* endloop (calculations of output layer) */
}

/*************************************************************
 * FUNCTION NAME: get_goals()                                *
 *                                                           *
 * DESCRIPTION:  read training data from a file.             *
 *************************************************************/
void get_goals()
{
   int j, k;
   FILE *f_ptr;

   #ifdef DEBUG
       printf("get goals function starting....\n");
   #endif
   /*---------------- OPEN TRAINING DATA FILE ------------------*/
   if ((f_ptr = fopen("letter_bip_trn.txt", "r")) == NULL) {
      fprintf(rpt_ptr, "\nThe input file vowel_data_trn_set could");
      fprintf(rpt_ptr, " not be opened. Terminating program.");
```

```
      exit(0);
   }/* endif */

   fscanf(f_ptr, "%d", &num_goals);
   fprintf(rpt_ptr, "\n The number of training samples is %d",
        num_goals);
   for(j=0; j < num_goals; j++){
      for(k=0; k < IN_UNTS; k++){
      fscanf(f_ptr, "%f", &input[j][k]);
      }/* endloop (# of input units) */

      /* set up threshold "neuron" */
      input[j][IN_UNTS] = 1.0;

      for(k=0; k < OUT_UNTS; k++){
       fscanf(f_ptr, "%f", &goals[j][k]);
      }/* endloop (# of target output units) */
   }/* endloop (# of training data sets) */
   fclose(f_ptr);
}

/*************************************************************
 * FUNCTION NAME: set_params()                               *
 *                                                           *
 * DESCRIPTION: initialize parameters before training        *
 *************************************************************/
void set_params()
{
   int j, k;

       #ifdef DEBUG
       printf("Set params function starting....\n");
       #endif

   fprintf(rpt_ptr, "\n\n\n------------------------------------------------");
   fprintf(rpt_ptr, "\nparameters are:\n\t\tlearning_rate = %f",
learning_rate);
   fprintf(rpt_ptr, "\n\t\tmomentum = %f", momentum);

   /* initialize delta_weights BEFORE training */
   for (j = 0; j < HDN_UNTS; j++){
     for (k = 0; k < IN_UNTS+1; k++){
        hid_layer[j].delta_weight[k] = 0.0;
     }/* weights on all inputs */
   }/* all of the hidden units */
   for (j = 0; j < OUT_UNTS; j++){
     for (k = 0; k < HDN_UNTS+1; k++){
       out_layer[j].delta_weight[k] = 0.0;
     }/* weights on all inputs */
   }/* all of the output units */
}

/*************************************************************
 * FUNCTION NAME: back_prop(i)                               *
 *                                                           *
 * DESCRIPTION: backpropagation learning.                    *
 *************************************************************/
```

```
void back_prop(i)
  int i;
{
   int j, k;
   float sum;

   tot_patt_err[i] = 0.0;

   for (j = 0; j < OUT_UNTS; j++){
      patt_err[i][j] = (goals[i][j] - out_layer[j].output)
                  *(goals[i][j] - out_layer[j].output);
      tot_patt_err[i] += patt_err[i][j];
      /* calc delta_weight (n+1) */
     /* out_layer[j].delta = (goals[i][j]-out_layer[j].output)* (1.0 -
out_layer[j].output)*out_layer[j].output;*/             /* binary sigmoid */
    /*  out_layer[j].delta = (goals[i][j] - out_layer[j].output) * 0.5 * (1 +
out_layer[j].output) * (1 - out_layer[j].output); */  /* bipolar sigmoid */
      /* out_layer[j].delta = (goals[i][j] - out_layer[j].output) * (1 +
out_layer[j].output) * (1 - out_layer[j].output); */ /* hyperbolic tangent */
    /* out_layer[j].delta = (goals[i][j] - out_layer[j].output) *
out_layer[j].output * -sumg;  */                    /* gaussian derivative */
     out_layer[j].delta = (goals[i][j]-out_layer[j].output) * (1.75 *
sin(1.75 * sumg) + sumg * cos(1.75 * sumg)) * exp(-(sumg * sumg)/2);   /*
morlet derivate */

      for (k = 0; k < HDN_UNTS+1; k++){
        out_layer[j].delta_weight[k] = learning_rate
                  * out_layer[j].delta * hid_layer[k].output
                  + (momentum * out_layer[j].delta_weight[k]);

      } /* Calculate all delta_weights feeding this output unit */
   } /* for all output units */

   for (j = 0; j < HDN_UNTS; j++){
     sum = 0.0;
     for (k = 0; k < OUT_UNTS; k++){
      sum += out_layer[k].delta*out_layer[k].weight[j];
     }
   /*  hid_layer[j].delta = sum* hid_layer[j].output*(1.0 -
hid_layer[j].output); */ /* binary sigmoid */
    /*  hid_layer[j].delta = sum * 0.5 * (1 + hid_layer[j].output) * (1 -
hid_layer[j].output); */  /* bipolar sigmoid */
    /*   hid_layer[j].delta = sum * (1 + hid_layer[j].output) * (1 -
hid_layer[j].output); */   /* hyperbolic tangent */
      hid_layer[j].delta = sum * -(1.75 * sin(1.75 * sumg) + sumg * cos(1.75
* sumg)) * exp(-(sumg * sumg)/2);       /* morlet wavelet  */
      for(k=0; k<IN_UNTS+1; k++){
        hid_layer[j].delta_weight[k] = learning_rate *hid_layer[j].delta*
input[i][k] + (momentum*hid_layer[j].delta_weight[k]);
       }/* Calculate delta_weights on all inputs */

   }/* for all hidden units */
/*************************************************************/
   /* ---------- NEW WEIGHTS -------------------- */
   for (j = 0; j < OUT_UNTS; j++){
       for (k = 0; k < HDN_UNTS+1; k++){
          out_layer[j].weight[k] =(out_layer[j].weight[k] +
```

```
                            out_layer[j].delta_weight[k]) ;
      } /* adjust all weights feeding this output unit */
   } /* for all output units */

   for (j = 0; j < HDN_UNTS; j++){
       for (k = 0; k < IN_UNTS+1; k++){
          hid_layer[j].weight[k] += hid_layer[j].delta_weight[k];
       }/* adjust weights on all inputs */
   }/* for all hidden units */
}

/*************************************************************
 * FUNCTION NAME: train()                                    *
 *                                                           *
 * DESCRIPTION:  Network Training.                           *
 *************************************************************/
void train()
{
 int j, iter = 0;
   #ifdef DEBUG
       printf("train function starting....\n");
   #endif
    do {
       sys_err =0.0;
       for (j=0; j < num_goals; j++){
         propagate(j);
         back_prop(j);
         sys_err += tot_patt_err[j];
       }/* for all target patterns */
       iter++;
       sys_err = 0.5*sys_err/num_goals;
       printf("j= %d  ssys_err = %f\n", iter, sys_err);
       if(sys_err < min_err_td){
           min_err_td = sys_err;
           min_err_iter = iter;
       }
    } while((sys_err > max_tot_err) && (iter < max_iterations));

    if (sys_err > max_tot_err){
       fprintf(rpt_ptr, "\nThe maximum number of iterations was");
       fprintf(rpt_ptr, " exceeded and the system failed to converge.");}
    else{
       fprintf(rpt_ptr, " \nThe system converged!!!!.");
    }
    fprintf(rpt_ptr, "\n iterations = %d, error = %g", iter, sys_err);
    fprintf(rpt_ptr, "\n\nThe minimum error was %g, and occurred",
            min_err_td);
    fprintf(rpt_ptr, " at the %d iteration", min_err_iter);
}

/*************************************************************
 * FUNCTION NAME: process()                                  *
 *                                                           *
 * DESCRIPTION: process test data.                           *
  *************************************************************/
void process()
{
```

```
   int i, j, k, correct= 0;
   float out_err;
   num_misses = 0;

   #ifdef DEBUG
       printf("process function starting....\n");
   #endif
   /*---------------- PROCESS TEST DATA ----------------------*/
   for (j=0; j < num_samples; j++){
       miss_flag = 0;
       propagate(j);

       for (k=0; k < OUT_UNTS; k++){
      out_err = known_output[j][k] - out_layer[k].output;
           if(out_err > 0.2){
            miss_flag++;
           }
     }/* endloop (# of output units) */
     /*------------ following line for stats --------- */
     if(miss_flag > 0) num_misses++;
   }/* endloop (# of input sets) */

   #ifdef DEBUG
       printf("evaluate function starting....\n");
   #endif
   /*---------------- check  classifications produced -------*/
   fprintf(rpt_ptr, "\n\n THE PERFORMANCE RESULTS FOR THIS RUN:");
   fprintf(rpt_ptr, "\n The number of testing samples in this run is %d",
        num_samples);
   fprintf(rpt_ptr, "\n The number of correct classifications is %d",
        (correct = num_samples - num_misses));
   fprintf(rpt_ptr, "\n The percentage of correct classifications is %f",
        (100.0*(float)correct)/((float)num_samples));
}

/************************************************************
 * FUNCTION NAME: main                                      *
 ************************************************************/
void main()
{
   void get_goals();
   void init_weights();
   void init_inputs();
   void process();
   void set_params();
   void train();

   int x,y;

   if ((rpt_ptr = fopen("crisp8h_letter_wavelet_tst_results.txt", "w+")) ==
NULL) {
      exit(0);
   }/* endif  */

   #ifdef DEBUG
       printf("Main function starting....\n");
   #endif
```

```
   general_init_weights();
   system("date > letter_bip_time");
   clock_t start = clock();                    // get initial clock

   for(lr = 1.0; lr < 10.0; lr += 1.0){
       learning_rate = lr/10.0;

      for(mom = 1.0; mom < 10.0; mom += 1.0){
         momentum = mom/10.0;
         min_err_td = 500.0;
         init_weights();
         set_params();
         get_goals();
         train();
         /*------------ SET WEIGHTS AND THRESHOLDS ---------*/
         init_inputs();
         process();

           /*fprintf(rpt_ptr, "\n the weights connected form input to hidden
layer are:");
           for(x= 0; x < HDN_UNTS; x++) {
               for(y = 0; y< IN_UNTS +1; y++)
                    fprintf(rpt_ptr, "\n %f", hid_layer[x].weight[y]);
           }
           fprintf(rpt_ptr, "\n the weights connected form hidden to output
layer are:");
           for(x= 0; x < OUT_UNTS; x++) {
              for(y = 0; y< HDN_UNTS +1; y++)
                    fprintf(rpt_ptr, "\n %f", out_layer[x].weight[y]);
           } */

      }
   }
   printf(" time elapsed: %f\n", ((double)clock() - start) / CLOCKS_PER_SEC);
/* record time when simulation stopped */
   system("date >> letter_bip_time");
   fclose(rpt_ptr);
}
```

Appendix B

DIVCON NEURON NETWORK SOURCE CODE

The main difference between the DN models is located in a function called propagate(). Therefore, the main source code is shown in the initial DN design and the other models will only contain their corresponding propagate() function.

B.1 Initial DN Design

```
// ------------------------------------------------------------------------
// File name:    neuron2.h
// Author:       Jovan Saenz
// Description: Initial DN model design with DNs in the hidden layer only.
// ------------------------------------------------------------------------

#include<cstdio>
#include<cstdlib>
#include<cmath>
#include<iostream>
#include<fstream>
#include <float.h>
using namespace std;

// inside each divcon neuron:
#define HDN_UNTS 2             // # of hidden neurons inside a divcon neuron
#define OUT_UNTS 1             // # of output neurons inside a divcon neuron
#define THRESH 1.0             // threshold for neurons
// ========================================================================
// nn using divcon neurons:
#define IN_UNTS 13             // number of inputs (app. dependent)
#define MX_INPTS 13            // max # of inputs (app. dependent)
#define FEED_IN 13            // max # of inputs to a neuron
#define HDN_NU 5              // # of hidden divcon neurons
#define OUT_NU 2              // # of output neurons
// ========================================================================
#define MX_OUTPTS 2              // max # of output neurons (app. dependent)
#define MX_PATTS 178             // max # of input patterns (app. dependent)
#define NET_THRESHOLD 1.0
#define MAX 1.0
#define MIN 0.01
#define MAX_ITER 3000            // max # of iterations
#define MAX_TOT_ERR 0.01         // max error tolerance

class neuron
{
  public:
    float weight[FEED_IN+1];            // weights for divcon neurons
    float delta_weight[FEED_IN+1];
    float delta;                        // delta value for divcon neurons
    float output;
```

```
    float innerweight[3];               // store inner weights plus the bias
    float inneroutput[3];               // inner neurons output plus the bias
    float innerdelta_weight[3];       //inner delta weights for inner neurons

    friend float random_normalized (int flag);
    friend void general_init_weights();
    friend void init_weights();
    friend void init_inputs();
    friend void propagate(int i);
    friend void get_goals();
    friend void set_params();
    friend void back_prop(int i);
    friend void train();
    friend void process();
};

neuron hid_layer[HDN_NU+1];
neuron out_layer[OUT_NU];
neuron general_hid_layer[HDN_NU+1];          // hold initial weights
neuron general_out_layer[OUT_NU];            // hold initial weights
static int num_samples, num_goals;
static float input[MX_PATTS][MX_INPTS+1];
static float known_output[MX_PATTS][MX_OUTPTS];
static float goals[MX_PATTS][MX_OUTPTS];
static float tot_patt_err[MX_PATTS];
static float patt_err[MX_PATTS][MX_OUTPTS];
static float learning_rate, momentum;
static float min_err_td;                // minimum error @ end of simulation
//static float sumg;                    // for gaussian and wavelet functions

// ============================================================================
// Description:  Random number generator function.
float random_normalized(int flag)
{
  float ret_val;
  ret_val=(float)rand()*(MAX-MIN)/(pow(2.0, 31.0)-1)+MIN;
  if(flag == 1)
    ret_val=ret_val*pow(-1.0,(double)rand());

  return (ret_val);
}
// ============================================================================
// ============================================================================
// Description: Initialize weights to random values and store them to provide
// constant initial weights for all variations in momentum and learning rate.
void general_init_weights()
{
  if(HDN_NU == 0)                   // if there are no hidden divcon neurons:
  {
    for(int j=0; j<OUT_NU; j++)               // store output neuron weights
    {
      for(int k=0; k<IN_UNTS + 1; k++)     // for all inputs to output neuron
        general_out_layer[j].weight[k] = random_normalized(1);

      for(int k=0; k<HDN_UNTS + 1; k++)          // store inner weights of DN
        general_out_layer[j].innerweight[k] = random_normalized(1);
    }
```

```
  }
  else                                // if hidden divcon neurons exist:
  {
    for(int j=0; j< HDN_NU; j++)      // initialize hidden layer weights of DN
    {
       for(int k=0; k<IN_UNTS +1; k++)    // for all inputs to each hidden DN
         general_hid_layer[j].weight[k] = random_normalized(1);

       for(int k=0; k<HDN_UNTS + 1; k++)          // store inner weights
         general_hid_layer[j].innerweight[k] = random_normalized(1);
    }

    for(int j=0; j<OUT_NU; j++)        // initialize output layer weights
    {
      for(int k=0; k<HDN_NU+1; k++)   // hidden DNs to each output neuron
        general_out_layer[j].weight[k] = random_normalized(1);
    }
  }
}
// ========================================================================
// ========================================================================
// Description: Initialize weights using the general weights stored.
void init_weights()
{
  int j, k;

  if(HDN_NU == 0)                                 // if there are no hidden DNs:
  {
    for(j=0; j<OUT_NU; j++)                       // for all output neurons:
    {
      for(k=0; k<IN_UNTS + 1; k++)  // inputs connected to each output neuron
        out_layer[j].weight[k] = general_out_layer[j].weight[k];

      for(k=0; k<HDN_UNTS+1; k++)        // for neurons inside each output DN
        out_layer[j].innerweight[k] = general_out_layer[j].innerweight[k];
    }
  }
  else                                       // if hidden divcon neurons exist:
  {
     hid_layer[HDN_NU].output = THRESH;        // set hidden layer threshold

     for(j=0; j<HDN_NU; j++)                              // for all hidden DNs
     {
       for(k=0; k<IN_UNTS+1; k++)        //inputs connected to each hidden DN
         hid_layer[j].weight[k] = general_hid_layer[j].weight[k];

       for(k=0; k< HDN_UNTS+1; k++)      // for neurons inside each hidden DN
         hid_layer[j].innerweight[k] = general_hid_layer[j].innerweight[k];
     }

     for(j=0; j<OUT_NU; j++)              // for all output neurons
     {
       for(k=0; k<HDN_NU+1; k++)          // hidden DNs to each output neuron
         out_layer[j].weight[k] = general_out_layer[j].weight[k];
     }
  }
}
```

```
// ============================================================================
//
============================================================================
// Description: read the test patterns from a file.
void init_inputs()
{
  int j, k;

  ifstream infile;            // open test file for reading
  infile.open("wine_bip_trn.txt");

  if (!infile)
  {
    cout << "Unable to open test file for reading...terminating program."<<
endl;
    exit(0);
  }

  infile >> num_samples;                  // load # of samples to be read
  for(j=0; j<num_samples; j++)
  {
    for(k=0; k<IN_UNTS; k++)
      infile >> input[j][k];              // read inputs

    input[j][IN_UNTS] = 1.0;              // set bias input

    for(k=0; k<OUT_NU; k++)
      infile >> known_output[j][k];       // read target outputs
  }
  infile.close();
}
// ============================================================================
// ============================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum2, fnet, fnet1, fnet2;
  float tem, tem2, tem3;                                  // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                                 // if there are no hidden DNs:
  {
    // ------- initialize and calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k] * input[i][k];      //accumulated sum
```

```
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];      //accumulated sum
      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1));  //inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                      // bias

      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ********************************************************************
      fnet = -(sum);
      out_layer[j].output = 1.0/(1.0 + exp(fnet));              // DN output
    }
  }
  else                                              // if hidden neurons exist:
  {
    // -------- initialize & calculate hidden divcon neurons responses:
    for(int j=0; j<HDN_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;

      // **** begin data processing inside each hidden divcon neuron:******
      for(int k=0; k<c; k++)
        sum1 += hid_layer[j].weight[k] * input[i][k];    // accumulated sum

      for(int k=c; k< IN_UNTS + 1; k++)
        sum2 += hid_layer[j].weight[k] * input[i][k];    // accumulated sum

      fnet1 = -(sum1);
      fnet2 = -(sum2);

     //  hid_layer[j].inneroutput[0]  =  1.0/(1.0  +  exp(fnet1));     //  inner
neuron 1 output (using binary sigmoid)
     //  hid_layer[j].inneroutput[1]  =  1.0/(1.0  +  exp(fnet2));     //  inner
neuron 2 output (using binary sigmoid)
      hid_layer[j].inneroutput[0] = 2.0/(1.0 + exp(fnet1)) - 1.0;  // bipolar
sigmoid
      hid_layer[j].inneroutput[1] = 2.0/(1.0 + exp(fnet2)) - 1.0;  // bipolar
sigmoid
        // hid_layer[j].inneroutput[0] = (exp(sum1) - exp(fnet1))/(exp(sum1)
+ exp(fnet1));   // hyberpolic tan
        // hid_layer[j].inneroutput[1] = (exp(sum2) - exp(fnet2))/(exp(sum2)
+ exp(fnet2));   // hyperbolic tan
     //   if(hid_layer[j].inneroutput[0]   !=  hid_layer[j].inneroutput[0])   {
// if a nan error is found...
       // hid_layer[j].inneroutput[0] = 1.0;
     // }
     //   if(hid_layer[j].inneroutput[1]   !=  hid_layer[j].inneroutput[1])   {
// if a nan error is found ...
       // hid_layer[j].inneroutput[1] = 1.0;
     // }
      //tem = ((sum1)/3.0) * ((sum1)/3.0);            // for wavelet function
      //tem2 = ((sum2)/3.0) * ((sum2)/3.0);           // for wavelet function
```

```
      //hid_layer[j].inneroutput[0] = cos(1.75 * ((sum1)/3.0)) * exp(-tem /
2);  // wavelet function - b=0,a=3
      //hid_layer[j].inneroutput[1] = cos(1.75 * ((sum2)/3.0)) * exp(-tem2 /
2);  // wavelet function - b=0,a=3
     //     hid_layer[j].inneroutput[0]     =     exp(-(sum1     *     sum1)/2);
// gaussian activation function
     //     hid_layer[j].inneroutput[1]     =     exp(-(sum2     *     sum2)/2);
// gaussian activation function
      hid_layer[j].inneroutput[2] = THRESH;

      for (int k=0; k<HDN_UNTS+1; k++)
        sum += hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k];
      // ***************************************************************

      fnet = -(sum)/NET_THRESHOLD;
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
      //tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
      //hid_layer[j].output   =   cos(1.75   *   ((sum)/3.0))   *   exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output        =        exp(-(sum        *        sum)/2);
// gaussian activation function
     // if(hid_layer[j].output != hid_layer[j].output) { // if a nan error is
found ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;

        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;

      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
      //tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
      //out_layer[j].output   =   cos(1.75   *   ((sum)/3.0))   *   exp(-tem3/2);
// wavelet function - b=0,a=3
     //      out_layer[j].output       =       exp(-(sum       *       sum)/2);
// gaussian activation function
      //sumg                          =                          ((sum)/3.0);
// for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {         // if nan
error is found:
       //  out_layer[j].output = 1.0;                      // modify output
     // }
    }
  }
}
// ===========================================================================
```

```
// ============================================================================
// Description: This function reads the training data from a file
void get_goals()
{
 int j, k;
  ifstream trnfile;

  trnfile.open("wine_bip_trn.txt");             // open training file
  if(!trnfile)
  {
    cout << "Unable to open training file... terminating program" << endl;
    exit(0);
  }
  trnfile >> num_goals;                  // store # of training patterns

  ofstream  fout("fiveNU_waveletb0a3_wine_tst_results.txt",  ios::app);    //
open results file for writing
  if(!fout)
  {
    std::cout  <<  "Unable  to  open  results  file...terminating  program."  <<
endl;
    exit(0);
  }

  fout << "\n The number of training samples is: " << num_goals;
  fout.close();

  for(j=0; j<num_goals; j++)
  {
     for(k=0; k<IN_UNTS; k++)
       trnfile >> input[j][k];           // store inputs

     input[j][IN_UNTS] = 1.0;            // set bias input

     for(k=0; k<OUT_NU; k++)
       trnfile >> goals[j][k];
  }
  trnfile.close();
}
// ============================================================================
// ============================================================================
// Description: initialize deltas
void set_params()
{
  int j, k;

  ofstream     fout("fiveNU_waveletb0a3_wine_tst_results.txt",     ios::app);
// open results file
  if(!fout)
  {
    std::cout  <<  "Unable  to  open  results  file...terminating  program."  <<
endl;
    exit(0);
  }

   fout << "\n\n\n----------------------------------------------\n";
   fout << "\nparameters are:\n\t\tlearning_rate = " << learning_rate;
```

```
   fout << "\n\t\tmomentum = " << momentum;
   fout.close();
  // initialize deltas before training begins:
  for(j=0; j<HDN_NU; j++)
  {
     for(k=0; k<IN_UNTS+1; k++)
       hid_layer[j].delta_weight[k] = 0.0;

     for(k=0; k<HDN_UNTS + 1; k++)
       hid_layer[j].innerdelta_weight[k] = 0.0;
  }

  if(HDN_NU == 0)                            // if there are no hidden DNs
  {
    for(j=0; j<OUT_NU; j++)
    {
      for(k=0; k<IN_UNTS + 1; k++)
        out_layer[j].delta_weight[k] = 0.0;
    }
  }
  else                                      // if hidden divcon neurons exist:
  {
    for(j=0; j<OUT_NU; j++)
    {
      for(k=0; k<HDN_NU+1; k++)
        out_layer[j].delta_weight[k] = 0.0;

    }
  }
}
// ============================================================================
// ============================================================================
// Description: This function implements the backpropagation learning.
void back_prop(int i)
{
  int j, k;
  float sum;

  tot_patt_err[i]=0.0;

  for(j=0; j<OUT_NU; j++)
  {
    patt_err[i][j] = (goals[i][j] - out_layer[j].output) * (goals[i][j] -
out_layer[j].output);
    tot_patt_err[i] += patt_err[i][j];

    // ----- calculate delta weights for output layer:
   //    out_layer[j].delta    =    (goals[i][j]-out_layer[j].output)*(1.0    -
out_layer[j].output)*out_layer[j].output;   // binary sigmoid derivative
    out_layer[j].delta  =  (goals[i][j]-out_layer[j].output)*  0.5  *  (1.0  +
out_layer[j].output)* (1 - out_layer[j].output);     // bipolar derivative
   //    out_layer[j].delta    =    (goals[i][j]-out_layer[j].output)*(1    +
out_layer[j].output) * (1 - out_layer[j].output);    // hyperbolic tan
    //out_layer[j].delta  =  -(goals[i][j]-out_layer[j].output)  *  (1.75  *
sin(1.75 * sumg) + sumg * cos(1.75 * sumg)) * exp(-(sumg * sumg)/2);   //
wavelet function -b=0,a=1
```

```
   //     out_layer[j].delta     =     (goals[i][j]-out_layer[j].output)     *
out_layer[j].output * -sumg;                                                  //
gaussian function

    if(HDN_NU == 0)
    {
      for(k=0; k<IN_UNTS + 1; k++)
        out_layer[j].delta_weight[k] = learning_rate * out_layer[j].delta *
input[i][k] + (momentum * out_layer[j].delta_weight[k]);
    }
    else
    {
      for(k=0; k<HDN_NU+1; k++)
        out_layer[j].delta_weight[k] = learning_rate * out_layer[j].delta *
hid_layer[k].output + (momentum * out_layer[j].delta_weight[k]);
    }
  }

  // ----- calculate the delta weights for hidden layer:
  for(j=0; j<HDN_NU; j++)
  {
    sum=0.0;
    for(k=0; k<OUT_NU; k++)
      sum += out_layer[k].delta * out_layer[k].weight[j];

   //   hid_layer[j].delta   =   sum   *   hid_layer[j].output   *   (1.0   -
hid_layer[j].output);  // binary sigmoid derivative
    hid_layer[j].delta = sum * 0.5 * (1 + hid_layer[j].output) * (1 -
hid_layer[j].output);  // bipolar sigmoid
   //  hid_layer[j].delta  =  sum  *  (1  +  hid_layer[j].output)  *  (1  -
hid_layer[j].output);    // hyperbolic tan derivative
     //  hid_layer[j].delta = sum * -(1.75 * sin(1.75 * sumg) + sumg *
cos(1.75 * sumg)) * exp(-(sumg * sumg)/2);           // wavelet function -
b=0,a=1
     //   hid_layer[j].delta   =   sum   *   hid_layer[j].output   *   -sumg;
// gaussian function
    for(k=0; k<IN_UNTS+1; k++)
      hid_layer[j].delta_weight[k] = learning_rate * hid_layer[j].delta *
input[i][k] + (momentum * hid_layer[j].delta_weight[k]);

    //       calculate       inner       neurons       delta       weights
************************************************************
    for(int x=0; x<HDN_UNTS + 1; x++)
      hid_layer[j].innerdelta_weight[x] = learning_rate * hid_layer[j].delta
*         hid_layer[j].inneroutput[x]         +         (momentum         *
hid_layer[j].innerdelta_weight[x]);
    //
************************************************************************
**********************
  }

  // calculate new weights for output layer:
  if(HDN_NU == 0)                          // if there are no hidden DNs:
  {
    for(j=0; j<OUT_NU; j++)
    {
      for(k=0; k<IN_UNTS + 1; k++)
```

```
        out_layer[j].weight[k]        =        (out_layer[j].weight[k]        +
out_layer[j].delta_weight[k]);

      // ***** calculate inner weights of neural unit ********************
      for(k=0; k<HDN_UNTS + 1; k++)
        out_layer[j].innerweight[k] += out_layer[j].delta_weight[k];
    }
  }
  else
  {
    for(j=0; j<OUT_NU; j++)
    {
         for(k=0; k<HDN_NU+1; k++)
           out_layer[j].weight[k]       =       (out_layer[j].weight[k]       +
out_layer[j].delta_weight[k]);
    }
  }

  // calculate new weights for hidden layer:
  for(j=0; j<HDN_NU; j++)
  {
    for(k=0; k<IN_UNTS+1; k++)
      hid_layer[j].weight[k] += hid_layer[j].delta_weight[k];
    // ***** calculate inner weights of neural unit **********************
    for(k=0; k<HDN_UNTS + 1; k++)
      hid_layer[j].innerweight[k] += hid_layer[j].innerdelta_weight[k];
    // ****************************************************************
  }
}
// ===========================================================================
// ===========================================================================
// Description: This function trains the neural network.
void train()
{
  int j, iter = 0;
  float sys_err;
  int min_err_iter;

  do {
      sys_err = 0.0;
      for(j=0; j<num_goals; j++)
      {
        propagate(j);             // propagate and backpropagate each pattern
        back_prop(j);
        sys_err += tot_patt_err[j];
      }
      iter++;
      sys_err = 0.5 * sys_err/num_goals;

      std::cout << "j= " << iter << "sys_err = " << sys_err << endl;
      if(sys_err<min_err_td)
      {
        min_err_td = sys_err;
        min_err_iter = iter;
      }
  } while((sys_err > MAX_TOT_ERR) && (iter<MAX_ITER));
```

```
  ofstream     fout("fiveNU_waveletb0a3_wine_tst_results.txt",     ios::app);
// open results file
  if(!fout)
  {
    std::cout  <<  "Unable  to  open  results  file...terminating  program."  <<
endl;
    exit(0);
  }

  if(sys_err > MAX_TOT_ERR)
     fout << endl << "The maximum number of iterations was exceeded and the
system failed to converge.";

  else
     fout << endl << "The system converged!!!!.";


   fout << endl << "iterations = " << iter << " error = " << sys_err;
   fout  << endl << endl << "The minimum error was " << min_err_td << " and
occurred at the " << min_err_iter << " iteration.";
   fout.close();
}
// ============================================================================
// ============================================================================
// Description: This function processes patterns through the network.
void process()
{
  int j, k, correct=0;
  float out_err;
  int num_misses;
  int miss_flag;
  float temp;

  num_misses =0;

  ofstream  misout("fiveNU_waveletb0a3_wine_tst_miss_report.txt",  ios::app);
// open report file
  if(!misout)
  {
    std::cout  <<  "Unable  to  open  report  file  ...  terminating  program."  <<
endl;
    exit(0);
  }
  misout << "For a learning rate =  " << learning_rate << endl;
  misout << "For a momentum =  " << momentum << endl;
  // ---------- process test data: ---------
  for(j=0; j<num_samples; j++)
  {
    miss_flag = 0;
    propagate(j);

    for(k=0; k<OUT_NU; k++)
    {
      out_err = known_output[j][k] - out_layer[k].output;
      if(out_err > 0.2)
        miss_flag++;
    }
```

```
    if (miss_flag > 0) num_misses++;
    // store pattern that failed to be recognized by the neural network:
    if(miss_flag > 0)             // if there was a miss:
    {
      misout << "The known pattern that could not be recognized was:" <<
"\t";
      for(k=0; k<IN_UNTS; k++)
      {
        misout << input[j][k] << "  ";
      }
      misout << endl;
    }

    if (miss_flag < 1)         // if correct recognition:
    {
       misout << "Target Pattern recognized correctly:" << "\t";
       for(k=0; k<OUT_NU; k++)
       {
          misout << known_output[j][k] << " " << out_layer[k].output << " ";
       }
    }
  }
  misout << endl;
  misout.close();

  ofstream     fout("fiveNU_waveletb0a3_wine_tst_results.txt",     ios::app);
// open results file
  if(!fout)
  {
    std::cout << "Unable to open results file...terminating program." <<
endl;
    exit(0);
  }
  // print results in a file:
   fout << endl << endl << "THE PERFORMANCE RESULTS FOR THIS RUN:" << endl;
   fout << "The number of testing samples  in this run is\t" << num_samples;
   correct = num_samples - num_misses;
   fout << endl <<" The number of correct classifications is\t" << correct;
   temp = (100.0 * (float)correct)/((float)num_samples);
   fout << endl << "The percentage of correct classifications is\t" << temp;
   fout.close();
}
// ============================================================================
// ----------------------------------------------------------------------------
// File name:  crispnn.cpp
// Author:     Jovan Saenz
//  this is the main program to simulate a MLP with DNs using the
// Backpropagation learning algorithm.
// ----------------------------------------------------------------------------

#include<cstdio>
#include<cmath>
#include<cstdlib>
#include<iostream>
#include<fstream>
#include"neuron2.h"
```

```
int main ()
{
   float lr, mom;

   general_init_weights();               // initialize random general weights
   system("date > vowel_trn_time");              // get current date and time
   clock_t start = clock();                          // get initial clock
   // start simulation for different values of learning rate and momentum:

   for(lr=1.0; lr<10.0; lr+=1.0)
     {
       learning_rate = lr/10.0;

       for (mom=1.0; mom<10.0; mom+=1.0)
       {
         momentum = mom/10.0;

         min_err_td = 500.0;
         init_weights();
         set_params();
         get_goals();
         train();
         init_inputs();
         process();
         std::cout << num_samples << endl;
       }
     }
   printf(" time elapsed: %f\n", ((double)clock() - start) / CLOCKS_PER_SEC);
/* record time when simulation stopped */
   system("date >> vowel_trn_time");
   return 0;
}
```

B.2 XOR Units Model

```
// ===========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum2, fnet, fnet1, fnet2;
  // float tem3;                            // for wavelet activation function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                       // if there are no hidden neural units:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
```

```
    sum=0.0;
    sum1=0.0;
    sum2=0.0;
    // ******* begin data processing inside each output divcon neuron:
    for(int k=0; k<c; k++)
      sum1 += out_layer[j].weight[k]*input[i][k]; // first inner neuron
    for(int k=c; k<IN_UNTS + 1; k++)
      sum2 += out_layer[j].weight[k]*input[i][k]; // second inner neuron

    fnet1 = (-sum1);
    fnet2 = (-sum2);

    out_layer[j].inneroutput[0]=1.0/(1.0+exp(fnet1));     // inner neuron 1
    out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
    out_layer[j].inneroutput[2] = THRESH;                 // bias

    for(int k=0; k<HDN_UNTS+1; k++)
      sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
    // ***************************************************************
    fnet = -(sum);
    out_layer[j].output = 1.0/(1.0 + exp(fnet));   // output divcon neuron
  }
}
else                                    // if hidden divcon neurons exist:
{
  // -------- initialize & calculate hidden divcon neurons responses:
  for(int j=0; j<HDN_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum2=0.0;
    // **** begin data processing inside each hidden divcon neuron:******
    for(int k=0; k<c; k++) {         // first half of inputs is processed
      // test if product of the input and weight is positive or negative:
      if ((hid_layer[j].weight[k]*input[i][k])>=0) { // weighted input is +
          if (hid_layer[j].weight[k] >= 1) {         // if weight is >= 1
             sum1 += 1.0;                            // excitatory signal = 1
          }
          else {                                     // if weight is < 1
             sum1 += -1.0;                           // inhibitory signal
          }
      }
      else {                             // if weighted input is negative:
          if ( (abs (hid_layer[j].weight[k])) >= 1) {   // if |weight| >= 1
             sum1 += -1.0;                           // inhibitory signal
          }
          else {                         // if magnitude of weight is < 1
             sum1 += 1.0;                // excitatory signal
          }
      }
    }
    for(int k=c; k< IN_UNTS + 1; k++) {  // second half of inputs processed
      // test product of the input and weight is positive or negative:
      if ((hid_layer[j].weight[k]*input[i][k])>=0) { // weighted input is +
         if (hid_layer[j].weight[k] >= 1) {          // if weight is >= 1
            sum2 += 1.0;                           // excitatory signal = 1
         }
```

```
            else {                                      // if weight is < 1
               sum2 += -1.0;                            // inhibitory signal
            }
         }
         else {                              // if weighted input is negative:
            if ( (abs (hid_layer[j].weight[k])) >= 1) {    // if |weight| >= 1
               sum2 += -1.0;                                // inhibitory signal
            }
            else {                                          // if |weight| is < 1
               sum2 += 1.0;                                 // excitatory signal
            }
         }
       }
       hid_layer[j].inneroutput[0] = sum1;          // output of inner neuron 1
       hid_layer[j].inneroutput[1] = sum2;          // output of inner neuron 2
       hid_layer[j].inneroutput[2] = THRESH;        // inner bias

       for (int k=0; k<HDN_UNTS+1; k++)
         sum  +=  hid_layer[j].innerweight[k]  *  hid_layer[j].inneroutput[k];
// get weighted input summation for perceptron
  // ***************************************************************
       fnet = -(sum)/NET_THRESHOLD;
      // hid_layer[j].output = 1.0/(1.0 + exp(fnet));         // binary sigmoid
       hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
      // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
      // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
      //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
       //hid_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
      // if(hid_layer[j].output != hid_layer[j].output) { // if nan error …
        //  hid_layer[j].output = 1.0;
      // }
     }
     // -------- initialize and calculate output neurons' responses:
     for(int j=0; j<OUT_NU; j++)
     {
       sum=0.0;
         for(int k=0; k<HDN_NU+1; k++)
           sum += out_layer[j].weight[k] * hid_layer[k].output;

       fnet = -(sum)/NET_THRESHOLD;
      //  out_layer[j].output = 1.0/(1.0 + exp(fnet));         // binary sigmoid
       out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
      // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
      // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
      //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      // out_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
      //                   sumg                  =                  ((sum)/3.0);
// for gaussian/wavelet function
      // if (out_layer[j].output != out_layer[j].output) {  // if nan error …
        //  out_layer[j].output = 1.0;
      // }
     }
  }
```

```
}
```

B.3 Shift Units Model

```
// ========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum2, fnet, fnet1, fnet2;
  // float tem3;                          // for wavelet activation function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                 // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                                 // if there are no hidden DNs:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k] * input[i][k];   // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0]=1.0/(1.0+exp(fnet1));     // inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                 // bias
      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ************************************************************
      fnet = -(sum);
      out_layer[j].output = 1.0/(1.0 + exp(fnet));              // output DN
    }
  }
  else                                                  // if hidden DNs exist:
  {
    // -------- initialize & calculate hidden divcon neurons responses:
    for(int j=0; j<HDN_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // **** begin data processing inside each hidden divcon neuron:******
      for(int k=0; k<c; k++) {          // first half of inputs processed
```

```
// test if the product of the input and weight is positive or negative:
   if ((hid_layer[j].weight[k]*input[i][k])>=0) {// weighted input is +
       if (hid_layer[j].weight[k] > 1) {            // if weight is > 1
          sum1 += input[i][k] * 2;                  // left shift
       }
       else {                                       // if weight is <= 1
            if (hid_layer[j].weight[k] == 1.0){     // if weight is = 1
                sum1 += input[i][k];                // no change to input
            }
            else {                                   // if weight is < 1
                sum1 += input[i][k] / 2;             // right shift
            }
       }
   }
   else {                              // if weighted input is negative:
       if ( (abs (hid_layer[j].weight[k])) > 1) {       // |weight| > 1
          sum1 += input[i][k] * 2;                      // left shift
       }
       else {                                       // |weight| is <= 1
            if ((abs(hid_layer[j].weight[k]))== 1.0) { // weight is = 1
               sum1 += input[i][k];                // no change to input
            }
            else {                                 // if weight is < 1
               sum1 += input[i][k] / 2;            // right shift
            }
       }
   }
 }
 for(int k=c; k< IN_UNTS + 1; k++) {  // second half of inputs processed
   // product of the input and weight is positive or negative:
   if ((hid_layer[j].weight[k]*input[i][k])>=0) { // weighted input is +
      if (hid_layer[j].weight[k] > 1) {           // if weight is > 1
         sum2 += input[i][k] * 2;                 // left shift
      }
      else {                                      // if weight is <= 1
           if (hid_layer[j].weight[k] == 1.0) {   // if weight is = 1
               sum2 += input[i][k];               // no change to input
           }
           else {                                 // if weight is < 1
               sum2 += input[i][k] / 2;           // right shift
           }
      }
   }
   else {                              // if weighted input is negative:
      if ( (abs (hid_layer[j].weight[k])) > 1) {       // |weight| > 1
         sum2 += input[i][k] * 2;                      // left shift
      }
      else {                                        // |weight| is <= 1
           if ((abs(hid_layer[j].weight[k]))==1.0) {   // if weight is 1
             sum2 += input[i][k];                 // no change to input
           }
           else {                                 // if weight is < 1
             sum2 += input[i][k] /                // right shift
           }
      }
   }
 }
```

```
      hid_layer[j].inneroutput[0] = sum1;                  // inner neuron 1
      hid_layer[j].inneroutput[1] = sum2;                  // inner neuron 2
      hid_layer[j].inneroutput[2] = THRESH;                // inner bias

      for (int k=0; k<HDN_UNTS+1; k++)
        sum  +=  hid_layer[j].innerweight[k]  *  hid_layer[j].inneroutput[k];
// get weighted input summation for perceptron
      // ************************************************************
      fnet = -(sum)/NET_THRESHOLD;
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     //        tem3        =        ((sum)/3.0)        *        ((sum)/3.0);
// for wavelet function
     //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // if(hid_layer[j].output != hid_layer[j].output) {  // if nan error ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // sumg = ((sum)/3.0);                 // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {   // if nan error …
       //  out_layer[j].output = 1.0;                       // modify output
     // }
    }
  }
}
```

B.4 Quantum Neurons Model

```
// ========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum1b, sum2, sum2b, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                        // for wavelet function
```

```
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                              // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                         // if even # of input neurons:
    c = IN_UNTS / 2;
  if(HDN_NU == 0)                              // if there are no hidden DNs:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k] * input[i][k];   // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                 // bias

      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ***************************************************************
      fnet = -(sum);
      out_layer[j].output = 1.0/(1.0 + exp(fnet));             // output DN
    }
  }
  else                                      // if hidden divcon neurons exist:
  {
    // -------- initialize & calculate hidden divcon neurons responses:
    for(int j=0; j<HDN_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum1b=0.0;
      sum2=0.0;
      sum2b=0.0;
      // **** begin data processing inside each hidden divcon neuron:*******
      for(int k=0; k<c; k++){
        sum1 += hid_layer[j].qweight[k][0] * input[i][k];    // sum for first
quantum neuron (first parameter)
        sum1b += hid_layer[j].qweight[k][1] * input[i][k];   // sum for first
quantum neuron (second parameter)
      }
      for(int k=c; k< IN_UNTS + 1; k++) {
        sum2  +=  hid_layer[j].qweight[k][0]  *  input[i][k];       //  sum  for
second quantum  neuron (first parameter)
        sum2b  +=  hid_layer[j].qweight[k][1]  *  input[i][k];     //  sum  for
second quantum neuron (second parameter)
      }
```

```
     // inner product of (sum1, sum1b) and (1,1) to generate output of first
quantum neuron:
      hid_layer[j].inneroutput[0]        =        sum1        +        sum1b;
// output of first quantum neuron
      // inner product of (sum2, sum2b) and (1,1) to generate output of
second quantum neuron:
      hid_layer[j].inneroutput[1]        =        sum2        +        sum2b;
// output of second quantum neuron

      hid_layer[j].inneroutput[2] = THRESH;
      for (int k=0; k<HDN_UNTS+1; k++)
        sum += hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k];
      // ***************************************************************
      fnet = -(sum)/NET_THRESHOLD;
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
     //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // if(hid_layer[j].output != hid_layer[j].output) {  // if nan error ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);       // gaussian function
     // sumg = ((sum)/3.0);                 // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {   // if nan error …
       //  out_layer[j].output = 1.0;                       // modify output
     // }
    }
  }
}
```

B.5 XOR & Shift Model

```
// ===========================================================================
// Description: This function propagates patterns through the Neural Network.
```

```
void propagate(int i)
{
  int c;
  float sum, sum1, sum2, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                              // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                    // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                              // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                                  // if there are no hidden DNs:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k] * input[i][k];   // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                 // bias

      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ****************************************************************
      fnet = -(sum);
      out_layer[j].output = 1.0/(1.0 + exp(fnet));              // output DN
    }
  }
  else                                                  // if hidden DNs exist:
  {
    // -------- initialize & calculate hidden divcon neurons responses:
    for(int j=0; j<HDN_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // **** begin data processing inside each hidden divcon neuron:******
      for(int k=0; k<c; k++){         // first half of inputs to be processed
     // test if the product of the input and weight is positive or negative:
        if ((hid_layer[j].weight[k]*input[i][k])>=0) { // weighted input +:
           if (hid_layer[j].weight[k] >= 1) {          // if weight is >= 1 :
               sum1 += 1.0;                          // excitatory signal
           }
           else {                                       // if weight is < 1:
               sum1 += -1.0;                         // inhibitory signal
           }
```

```
        }
        else {                                  // if weighted input is negative:
           if ( (abs (hid_layer[j].weight[k])) >= 1) {      // |weight| >= 1:
               sum1 += -1.0;                                // inhibitory signal
           }
           else {                                           // |weight| < 1:
               sum1 += 1.0;                                 // excitatory signal
           }
        }
      //  sum1  +=  input[i][k]  *  hid_layer[j].weight[k];     //  traditional
weighted input summation
      }
      for(int k=c; k< IN_UNTS + 1; k++) {  // second half of inputs processed
      // test if the product of the input and weight is positive or negative:
        if ( (hid_layer[j].weight[k]*input[i][k])>=0) { // weighted input +
           if (hid_layer[j].weight[k] > 1) {            // if weight is > 1:
              sum2 += input[i][k] * 2;                  // left shift
           }
           else {                                       // if weight is <= 1:
              if (hid_layer[j].weight[k] == 1.0) {      // if weight is = 1:
                 sum2 += input[i][k];                   // no change to input
              }
              else {                                    // if weight is < 1:
                 sum2 += input[i][k] / 2;               // right shift
              }
           }
        }
        else {                                  // if weighted input is negative:
             if ( (abs (hid_layer[j].weight[k])) > 1) {      // |weight| > 1:
                 sum2 += input[i][k] * 2;                    // left shift
             }
             else {                                         // |weight| <= 1:
                 if ((abs(hid_layer[j].weight[k]))==1.0) { // weight is 1:
                     sum2 += input[i][k];               // no change to input
                 }
                 else {                                 // if weight is < 1:
                      sum2 += input[i][k] / 2;          // right shift
                 }
             }
        }
    //  sum2+=hid_layer[j].weight[k]*input[i][k]; // weighted input summation
      }

     hid_layer[j].inneroutput[0] = sum1;      // output of xor unit
     hid_layer[j].inneroutput[1] = sum2;      // output of shift unit
     hid_layer[j].inneroutput[2] = THRESH;
      for (int k=0; k<HDN_UNTS+1; k++)
        sum += hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k];
      // ************************************************************
      fnet = -(sum)/NET_THRESHOLD;
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
    //  hid_layer[j].output  =  (2.0/(1.0  +  exp(fnet))  -  1.0)  +  (2.0/(1.0  +
exp(fnet + 3)) - 1.0) + (2.0/(1.0 + exp(fnet + 6)) - 1.0); // bipolar sigmoid
     //hid_layer[j].output    =    2.0/(1.0    +    exp(fnet/0.5))    -    1.0;
// bipolar sigmoid with threshold
      hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
```

```
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
     //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // if(hid_layer[j].output != hid_layer[j].output) {    // nan error ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
     //out_layer[j].output  =  (2.0/(1.0  +  exp(fnet))  -  1.0)  +  (2.0/(1.0  +
exp(fnet + 3)) - 1.0) + (2.0/(1.0 + exp(fnet + 6)) - 1.0); // bipolar sigmoid
     //   out_layer[j].output   =   2.0/(1.0   +   exp(fnet/0.5))   -   1.0;
// bipolar sigmoid with threshold
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // sumg = ((sum)/3.0);                  // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {      // nan error …
       //  out_layer[j].output = 1.0;
     // }
    }
  }
}
```

B.6 XOR & Quantum Model

```
// ===========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum2, sum2b, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                              // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                    // if there are no hidden divcon neurons:
  {
```

```
  // ------- initialize and  calculate the output of the divcon neuron:
  for(int j=0; j<OUT_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum2=0.0;
    // ******* begin data processing inside each output divcon neuron:
    for(int k=0; k<c; k++)
      sum1 += out_layer[j].weight[k] * input[i][k];   // first inner neuron
    for(int k=c; k<IN_UNTS + 1; k++)
      sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

    fnet1 = (-sum1);
    fnet2 = (-sum2);

    out_layer[j].inneroutput[0]=1.0/(1.0 + exp(fnet1)); // neuron 1 output
    out_layer[j].inneroutput[1]=1.0/(1.0 + exp(fnet2)); // neuron 2 output
    out_layer[j].inneroutput[2] = THRESH;               // bias

    for(int k=0; k<HDN_UNTS+1; k++)
      sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
    // ************************************************************
    fnet = -(sum);
    out_layer[j].output = 1.0/(1.0 + exp(fnet));      // output of DN
  }
}
else                                          // if hidden DNs exit:
{
  // -------- initialize & calculate hidden divcon neurons' responses:
  for(int j=0; j<HDN_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum2=0.0;
    sum2b=0.0;
    // **** begin data processing inside each hidden divcon neuron:******
    // half of the inputs are processed by the XOR operation unit:
    for(int k=0; k<c; k++){
   // test if the product of the input and weight is positive or negative:
      if ((hid_layer[j].weight[k]*input[i][k])>=0) { // if weighted input +
         if (hid_layer[j].weight[k] >= 1) {          // if weight is >= 1:
            sum1 += 1.0;                           // excitatory signal = 1
         }
         else {                                      // if weight is < 1:
            sum1 += -1.0;                          // inhibitory signal
         }
      }
      else {                                 // if weighted input is negative:
         if ( (abs (hid_layer[j].weight[k])) >= 1) {     // |weight| >= 1:
            sum1 += -1.0;                               // inhibitory signal
         }
         else {                                          // |weight| is < 1:
            sum1 += 1.0;                                 // excitatory signal
         }
      }
    }
    // the remaining half of the inputs are processed by a quantum neuron:
```

```
      for(int k=c; k< IN_UNTS + 1; k++) {
        sum2  +=  hid_layer[j].qweight[k][0]  *  input[i][k];       //  sum  for
second quantum  neuron (first parameter)
        sum2b  +=  hid_layer[j].qweight[k][1]  *  input[i][k];     //  sum  for
second quantum neuron (second parameter)
      }
     // inner product of (sum1, sum1b) and (1,1) to generate output of first
quantum neuron:
      hid_layer[j].inneroutput[0] = sum1;           // output of XOR unit
      // inner product of (sum2, sum2b) and (1,1) to generate output of
second quantum neuron:
      hid_layer[j].inneroutput[1] = sum2 + sum2b; // output of quantum neuron
      hid_layer[j].inneroutput[2] = THRESH;

      for (int k=0; k<HDN_UNTS+1; k++)
        sum += hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k];
      // ***************************************************************
      fnet = -(sum)/NET_THRESHOLD;
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
     //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output = exp(-(sum * sum)/2);       // gaussian function
     // if(hid_layer[j].output != hid_layer[j].output) {     // nan error ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);    // gaussian function
     // sumg = ((sum)/3.0);                  // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {      // nan error …
       //  out_layer[j].output = 1.0;
     // }
    }
  }
}
```

B.7 Shift & Quantum Model

```
// ============================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum1b, sum2, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                              // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                                 // if there are no hidden DNs:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k] * input[i][k];   // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                 // bias

      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ***************************************************************
      fnet = -(sum);
      out_layer[j].output = 1.0/(1.0 + exp(fnet));        // output of DN
    }
  }
  else                                                  // if hidden DNs exist:
  {
    // -------- initialize & calculate hidden divcon neurons' responses:
    for(int j=0; j<HDN_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum1b=0.0;
      sum2=0.0;
     // sum2b=0.0;
      // **** begin data processing inside each hidden divcon neuron:******
      // half of the inputs are processed:
      for(int k=0; k<c; k++){
     // test if the product of the input and weight is positive or negative:
          sum1     +=     hid_layer[j].qweight[k][0]     *     input[i][k];
// sum for first quantum neuron (state 0)
```

```
          sum1b     +=     hid_layer[j].qweight[k][1]     *     input[i][k];
// sum for first quantum neuron (state 1)
      }
      // the remaining half of the inputs are processed:
      for(int k=c; k< IN_UNTS + 1; k++) {
     // test if the product of the input and weight is positive or negative:
        if ((hid_layer[j].weight[k]*input[i][k])>=0) { // weighted input +:
           if (hid_layer[j].weight[k] > 1) {           // if weight is > 1:
              sum2 += input[i][k] * 2;                 // left shift
           }
           else {                                      // if weight is <= 1:
              if (hid_layer[j].weight[k] == 1.0) {     // if weight is = 1:
                 sum2 += input[i][k];                  // no change to input
              }
              else {                                   // if weight is < 1:
                 sum2 += input[i][k] / 2;              // right shift
              }
           }
        }
        else {                             // if weighted input is negative:
          if ( (abs (hid_layer[j].weight[k])) > 1) {        // |weight| > 1
             sum2 += input[i][k] * 2;                       // left shift
          }
          else {                                           // |weight| <= 1:
             if ( (abs (hid_layer[j].weight[k])) == 1.0) {  // weight is = 1
                sum2 += input[i][k];                  // no change to input
             }
             else {                                   // if weight is < 1:
                sum2 += input[i][k] / 2;              // right shift
             }
          }
        }
      }
      hid_layer[j].inneroutput[1] = sum2;           // output of shift unit
      hid_layer[j].inneroutput[0] = sum1 + sum1b; // output of quantum neuron
      hid_layer[j].inneroutput[2] = THRESH;

      for (int k=0; k<HDN_UNTS+1; k++){
        sum  +=  hid_layer[j].innerweight[k]  *  hid_layer[j].inneroutput[k];
// weighted sum for perceptron output of DN
      }
      // ************************************************************
      fnet                        =                        -(sum)/NET_THRESHOLD;
// for traditional output perceptron
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output = exp(-(sum * sum)/2        // gaussian function
     // if(hid_layer[j].output != hid_layer[j].output) {     // nan error ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
```

```
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);       // gaussian function
     // sumg = ((sum)/3.0);                   // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {     // nan error …
       //  out_layer[j].output = 1.0;
     // }
    }
  }
}
```

B.8 2XOR & Quantum Model

```
// ===========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum1b, sum2, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                          // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                    // if there are no hidden divcon neurons:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k] * input[i][k];   // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);
```

```
    out_layer[j].inneroutput[0]=1.0/(1.0 + exp(fnet1));   // inner neuron 1
    out_layer[j].inneroutput[1]=1.0/(1.0 + exp(fnet2));   // inner neuron 2
    out_layer[j].inneroutput[2] = THRESH;                 // bias

    for(int k=0; k<HDN_UNTS+1; k++)
      sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
    // ************************************************************
    fnet = -(sum);
    out_layer[j].output = 1.0/(1.0 + exp(fnet));           // output of DN
  }
}
else                                      // if hidden divcon neurons exist:
{
  // -------- initialize & calculate hidden divcon neurons' responses:
  for(int j=0; j<HDN_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum1b=0.0;
    sum2=0.0;
    // **** begin data processing inside each hidden divcon neuron:*******
    // half of the inputs are processed:
    for(int k=0; k<c; k++){
   // test if the product of the input and weight is positive or negative:
      if ((hid_layer[j].weight[k]*input[i][k]) >= 0) {// weighted input +
         if (hid_layer[j].weight[k] >= 1) {           // if weight is > 1:
            sum1 += 1.0;                             // excitatory signal = 1
         }
         else {                                        // if weight is <= 1:
            sum1 += -1.0;                              // inhibitory signal
         }
      }
      else {                              // if weighted input is negative:
         if ( (abs (hid_layer[j].weight[k])) >= 1) {   // |weight| is >= 1:
            sum1 += -1.0;                              // inhibitory signal
         }
         else {                                        // |weight| is <= 1:
            sum1 += 1.0;                               // excitatory signal
         }
      }
    }
    // the remaining half of the inputs are processed:
    for(int k=c; k< IN_UNTS + 1; k++) {
    // test if the product of the input and weight is positive or negative:
       if ((hid_layer[j].weight[k]*input[i][k])>=0) { // weighted input +
          if (hid_layer[j].weight[k] >= 1) {           // if weight is > 1:
             sum2 += 1.0;                           // excitatory signal = 1
          }
          else {                                      // if weight is <= 1:
              sum2 += -1.0;                           // inhibitory signal
          }
       }
       else {                             // if weighted input is negative:
         if ( (abs (hid_layer[j].weight[k])) >= 1) {    // |weight| is > 1:
            sum2 += -1.0;                               // inhibitory signal
         }
         else {                                         // |weight| is <= 1:
```

```
              sum2 += 1.0;                                  // excitatory signal
            }
          }
        }
        hid_layer[j].inneroutput[0] = sum1;          // output of first XOR unit
        hid_layer[j].inneroutput[1] = sum2;         // output of second XOR unit
        hid_layer[j].inneroutput[2] = THRESH;

        for (int k=0; k<HDN_UNTS+1; k++){
         //  sum  +=  hid_layer[j].innerweight[k]  *  hid_layer[j].inneroutput[k];
// weighted sum for perceptron output of DN
          sum   +=   hid_layer[j].qweight[k][0]   *   hid_layer[j].inneroutput[k];
// weighted sum for quantum  neuron (state 0)
          sum1b  +=  hid_layer[j].qweight[k][1]  *  hid_layer[j].inneroutput[k];
// weighted sum for quantum neuron  (state 1)

        }
        // ***************************************************************
        hid_layer[j].output = sum + sum1b;          // output of quantum neuron
      }
      // -------- initialize and calculate output neurons' responses:
      for(int j=0; j<OUT_NU; j++)
      {
        sum=0.0;
          for(int k=0; k<HDN_NU+1; k++)
            sum += out_layer[j].weight[k] * hid_layer[k].output;
        fnet = -(sum)/NET_THRESHOLD;
       //  out_layer[j].output = 1.0/(1.0 + exp(fnet));        // binary sigmoid
        out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
       // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
       // tem3 = ((sum)/3.0) * ((sum)/3.0);              // for wavelet function
       //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
       // out_layer[j].output = exp(-(sum * sum)/2);       // gaussian function
       // sumg = ((sum)/3.0);                   // for gaussian/wavelet function
       // if (out_layer[j].output != out_layer[j].output) { // nan error …
         //  out_layer[j].output = 1.0;                   // modify output
       // }
      }
    }
}
```

B.9 2Shift & Quantum Model

```
// ===================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum1b, sum2, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                              // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                               // if odd # of input neurons:
```

```
  c = (IN_UNTS / 2) +1;
else                                        // if even # of input neurons:
  c = IN_UNTS / 2;

if(HDN_NU == 0)                             // if there are no HDs neurons:
{
  // ------- initialize and  calculate the output of the divcon neuron:
  for(int j=0; j<OUT_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum2=0.0;
    // ******* begin data processing inside each output divcon neuron:
    for(int k=0; k<c; k++)
      sum1 += out_layer[j].weight[k] * input[i][k];   // first inner neuron
    for(int k=c; k<IN_UNTS + 1; k++)
      sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

    fnet1 = (-sum1);
    fnet2 = (-sum2);

    out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
    out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
    out_layer[j].inneroutput[2] = THRESH;                 // bias

    for(int k=0; k<HDN_UNTS+1; k++)
      sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
    // ************************************************************
    fnet = -(sum);
    out_layer[j].output = 1.0/(1.0 + exp(fnet));          // output of DN
  }
}
else                                              // if hidden DN exist:
{
  // -------- initialize & calculate hidden divcon neurons' responses:
  for(int j=0; j<HDN_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum1b=0.0;
    sum2=0.0;
    // **** begin data processing inside each hidden divcon neuron:******
    // half of the inputs are processed:
    for(int k=0; k<c; k++){
   // test if the product of the input and weight is positive or negative:
      if ((hid_layer[j].weight[k]*input[i][k]) >= 0) { // weighted input +
         if (hid_layer[j].weight[k] > 1) {           // if weight is > 1:
            sum1 += input[i][k] * 2;                 // left shift
         }
         else {                                      // if weight is <= 1:
            if (hid_layer[j].weight[k] == 1.0) {     // if weight is = 1
                sum1 += input[i][k];                 // no change to input
            }
            else {                                   // if weight is < 1:
                sum1 += input[i][k] / 2              // right shift
            }
         }
```

```
        }
        else {                              // if weighted input is negative:
           if ( (abs (hid_layer[j].weight[k])) > 1) {       // |weight| >= 1
              sum1 += input[i][k] * 2;                      // left shift
           }
           else {                                           // |weight| <= 1:
              if ( (abs (hid_layer[j].weight[k])) == 1.0) { // weight is = 1
                 sum1 += input[i][k];                    // no change to input
              }
              else {                                        // if weight is < 1:
                 sum1 += input[i][k] / 2;                 // right shift
              }
           }
        }
       }
       // the remaining half of the inputs are processed:
       for(int k=c; k< IN_UNTS + 1; k++) {
      // test if the product of the input and weight is positive or negative:
          if ((hid_layer[j].weight[k]*input[i][k]) >= 0) { // weighted input +
             if (hid_layer[j].weight[k] > 1) {           // if weight is > 1:
                sum2 += input[i][k] * 2;                 // left shift
             }
             else {                                      // if weight is <= 1:
                if (hid_layer[j].weight[k] == 1.0) {     // if weight is = 1:
                   sum2 += input[i][k];                  // no change to input
                }
                else {                                   // if weight is < 1:
                   sum2 += input[i][k] / 2;              // right shift
                }
             }
          }
          else {                            // if weighted input is negative:
            if ( (abs (hid_layer[j].weight[k])) > 1) {       // |weight| > 1:
               sum2 += input[i][k] * 2;                     // left shift
            }
            else {                                          // |weight| <= 1
               if ((abs (hid_layer[j].weight[k]))==1.0) { // if weight is = 1
                  sum2 += input[i][k];                  // no change to input
               }
               else {                                    // if weight is < 1:
                  sum2 += input[i][k] / 2;               // right shift
               }
            }
          }
       }

       hid_layer[j].inneroutput[0] = sum1;       // output of first shift unit
       hid_layer[j].inneroutput[1] = sum2;      // output of second shift unit
       hid_layer[j].inneroutput[2] = THRESH;

       for (int k=0; k<HDN_UNTS+1; k++){
        //  sum  +=  hid_layer[j].innerweight[k]  *  hid_layer[j].inneroutput[k];
// weighted sum for perceptron output of DN
         sum   +=   hid_layer[j].qweight[k][0]   *   hid_layer[j].inneroutput[k];
// weighted sum for quantum  neuron (state 0)
         sum1b  +=  hid_layer[j].qweight[k][1]  *  hid_layer[j].inneroutput[k];
// weighted sum for quantum neuron  (state 1)
```

```
      }
      // ****************************************************************
      hid_layer[j].output = sum + sum1b;          // output of quantum neuron
     // fnet = -(sum)/NET_THRESHOLD;     // for traditional output perceptron
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
     // hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0; // bipolar sigmoid
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output = exp(-(sum * sum)/2);     // gaussian function
     // if(hid_layer[j].output != hid_layer[j].output) {     // nan error ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // sumg = ((sum)/3.0);                 // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {      // nan error …
       //  out_layer[j].output = 1.0;
     // }
    }
  }
}
```

B.10 2Quantum & XOR Model

```
// ========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum1b, sum2, sum2b, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                          // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;
  if(HDN_NU == 0)                    // if there are no hidden divcon neurons:
```

```
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k]*input[i][k];    // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                 // bias

      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ************************************************************
      fnet = -(sum);
      out_layer[j].output = 1.0/(1.0 + exp(fnet));          // output of DN
    }
  }
  else                                      // if hidden divcon neurons exist:
  {
    // -------- initialize & calculate hidden divcon neurons' responses:
    for(int j=0; j<HDN_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum1b=0.0;
      sum2=0.0;
      sum2b=0.0;
      // **** begin data processing inside each hidden divcon neuron:******
      // half of the inputs are processed:
      for(int k=0; k<c; k++){
          sum1      +=      hid_layer[j].qweight[k][0]      *      input[i][k];
// sum for first quantum neuron (state 0)
          sum1b     +=      hid_layer[j].qweight[k][1]      *      input[i][k];
// sum for first quantum neuron (state 1)
      }
      // the remaining half of the inputs are processed:
      for(int k=c; k< IN_UNTS + 1; k++) {

           sum2  +=  hid_layer[j].qweight[k][0]  *  input[i][k];      //  sum  for
second quantum  neuron (state 0)
           sum2b  +=  hid_layer[j].qweight[k][1]  *  input[i][k];     //  sum  for
second quantum neuron (state 1)
      }

        hid_layer[j].inneroutput[0]=sum1+sum1b; // first quantum neuron
      // second quantum neuron:
        hid_layer[j].inneroutput[1]=sum2+sum2b;      // second quantum neuron
```

```
        hid_layer[j].inneroutput[2] = THRESH;

      for (int k=0; k<HDN_UNTS+1; k++){
      // test if the product of the input and weight is positive or negative:
          if ( (hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k]) >=
0) {    // if weighted input is positive:
              if (hid_layer[j].innerweight[k] >= 1) {    // if weight is > 1:
                 sum += 1.0;                           // excitatory signal = 1
              }
              else {                                   // if weight is <= 1:
                 sum += -1.0;                          // inhibitory signal
              }
          }
          else {                              // if weighted input is negative:
               if ((abs (hid_layer[j].innerweight[k]))>=1) { // |weight| > 1:
                   sum += -1.0;                            // inhibitory signal
               }
               else {                          // if magnitude of weight is < 1
                   sum += 1.0;                             // excitatory signal
               }
          }
      }
      // ************************************************************
        hid_layer[j].output = sum;                          // output of XOR unit
     //               fnet               =                -(sum)/NET_THRESHOLD;
// for traditional output perceptron
     // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
    //  hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0; // bipolar sigmoid
     // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      //hid_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // if(hid_layer[j].output != hid_layer[j].output) {     // nan error ...
       //  hid_layer[j].output = 1.0;
     // }
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);     // gaussian function
     // sumg = ((sum)/3.0);                  // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) { // nan error …
       //  out_layer[j].output = 1.0;
     // }
```

```
    }
  }
}
```

B.11 2Quantum & Shift Model

```
// ========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum1b, sum2, sum2b, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                          // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                                  // if there are no hidden DNs:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k]*input[i][k];    // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k] * input[i][k];  // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                 // bias

      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ************************************************************
      fnet = -(sum);
      out_layer[j].output = 1.0/(1.0 + exp(fnet));       // output of DN
    }
  }
  else                                        // if hidden divcon neurons exist:
  {
    // -------- initialize & calculate hidden divcon neurons' responses:
    for(int j=0; j<HDN_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
```

```
      sum1b=0.0;
      sum2=0.0;
      sum2b=0.0;
      // **** begin data processing inside each hidden divcon neuron:****
      // half of the inputs are processed:
      for(int k=0; k<c; k++){

          sum1      +=      hid_layer[j].qweight[k][0]      *      input[i][k];
// sum for first quantum neuron (state 0)
          sum1b     +=      hid_layer[j].qweight[k][1]      *      input[i][k];
// sum for first quantum neuron (state 1)
      }
      // the remaining half of the inputs are processed:
      for(int k=c; k< IN_UNTS + 1; k++) {

           sum2 += hid_layer[j].qweight[k][0] * input[i][k];      // sum for
second quantum  neuron (state 0)
           sum2b += hid_layer[j].qweight[k][1] * input[i][k];    // sum for
second quantum neuron (state 1)
      }

        hid_layer[j].inneroutput[0] = sum1 + sum1b;   // first quantum neuron
      // inner product of (sum2, sum2b) and (1,1) to generate output of
second quantum neuron:
        hid_layer[j].inneroutput[1] = sum2 + sum2b;  // second quantum neuron
        hid_layer[j].inneroutput[2] = THRESH;

      for (int k=0; k<HDN_UNTS+1; k++){
      // test if the product of the input and weight is positive or negative:
          if ( (hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k]) >=
0) {    // if weighted input is positive:
              if (hid_layer[j].innerweight[k] > 1) {     // if weight is > 1:
                 sum += hid_layer[j].inneroutput[k] * 2;        // left shift
              }
              else {                                        // if weight is <= 1:
                 if (hid_layer[j].weight[k] == 1.0) {    // if weight is = 1
                    sum += hid_layer[j].inneroutput[k]; // no change to input
                 }
                 else {                                 // if weight is < 1:
                    sum += hid_layer[j].inneroutput[k] / 2;    // right shift
                 }
              }
          }
          else {                            // if weighted input is negative:
               if ( (abs (hid_layer[j].innerweight[k])) > 1) {// |weight| > 1
                  sum += hid_layer[j].inneroutput[k] * 2;       // left shift
               }
               else {                                             // |weight| < 1
                  if ((abs(hid_layer[j].innerweight[k]))==1.0) {// weight = 1
                      sum+=hid_layer[j].inneroutput[k]; // no change to input
                  }
                  else {                                  // if weight is < 1
                      sum += hid_layer[j].inneroutput[k] / 2;  // right shift
                  }
               }
          }
      }
```

```
      // ************************************************************
        hid_layer[j].output = sum;                    // output of shift unit
      //               fnet              =               -(sum)/NET_THRESHOLD;
// for traditional output perceptron
      // hid_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      // hid_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0; // bipolar sigmoid
      // hid_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
      // tem3 = ((sum)/3.0) * ((sum)/3.0);             // for wavelet function
      //  hid_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
       //hid_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
      // if(hid_layer[j].output != hid_layer[j].output) {     // nan error ...
        //  hid_layer[j].output = 1.0;
      // }
     }
     // -------- initialize and calculate output neurons' responses:
     for(int j=0; j<OUT_NU; j++)
     {
       sum=0.0;
         for(int k=0; k<HDN_NU+1; k++)
           sum += out_layer[j].weight[k] * hid_layer[k].output;
       fnet = -(sum)/NET_THRESHOLD;
      //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
       out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
      // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
      // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
      //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      // out_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
      // sumg = ((sum)/3.0);                  // for gaussian/wavelet function
      // if (out_layer[j].output != out_layer[j].output) {      // nan error …
        //  out_layer[j].output = 1.0;
      // }
     }
  }
}
```

B.12 3-XOR Model

```
// ===========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum2, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                          // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;
  if(HDN_NU == 0)                    // if there are no hidden divcon neurons:
```

```
{
  // ------- initialize and  calculate the output of the divcon neuron:
  for(int j=0; j<OUT_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum2=0.0;
    // ******* begin data processing inside each output divcon neuron:
    for(int k=0; k<c; k++)
      sum1 += out_layer[j].weight[k]*input[i][k]; // first inner neuron
    for(int k=c; k<IN_UNTS + 1; k++)
      sum2 += out_layer[j].weight[k]*input[i][k];  // second inner neuron

    fnet1 = (-sum1);
    fnet2 = (-sum2);

    out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
    out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
    out_layer[j].inneroutput[2] = THRESH;                 // bias

    for(int k=0; k<HDN_UNTS+1; k++)
      sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
    // ************************************************************
    fnet = -(sum);
    out_layer[j].output = 1.0/(1.0 + exp(fnet));           // output of DN
  }
}
else                                      // if hidden divcon neurons exist:
{
  // -------- initialize & calculate hidden divcon neurons' responses:
  for(int j=0; j<HDN_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum2=0.0;

    // **** begin data processing inside each hidden divcon neuron ****
    // half of the inputs are processed:
    for(int k=0; k<c; k++){
   // test if the product of the input and weight is positive or negative:
      if ( (hid_layer[j].weight[k]*input[i][k]) >= 0) { // weighted input +
         if (hid_layer[j].weight[k] >= 1) {          // if weight is >= 1:
            sum1 += 1.0;                            // excitatory signal = 1
         }
         else {                                        // if weight is <= 1:
            sum1 += -1.0;                              // inhibitory signal
         }
      }
      else {                              // if weighted input is negative:
         if ( (abs (hid_layer[j].weight[k])) >= 1) {     // |weight| >= 1:
            sum1 += -1.0;                                // inhibitory signal
         }
         else {                                            // |weight| < 1:
            sum1 += 1.0;                                // excitatory signal
         }
      }
   }
```

```
      // the remaining half of the inputs are processed:
      for(int k=c; k< IN_UNTS + 1; k++) {
      // test if the product of the input and weight is positive or negative:
         if ( (hid_layer[j].weight[k]*input[i][k]) >= 0) {// weighted input +
            if (hid_layer[j].weight[k] >= 1) {           // if weight is > 1:
               sum2 += 1.0;                            // excitatory signal = 1
            }
            else {                                      // if weight is <= 1:
                sum2 += -1.0;                           // inhibitory signal
            }
         }
         else {                               // if weighted input is negative:
           if ( (abs (hid_layer[j].weight[k])) >= 1) {      // |weight| > 1:
              sum2 += -1.0;                                 // inhibitory signal
           }
           else {                                           // |weight| <= 1:
              sum2 += 1.0;                                  // excitatory signal
           }
         }
      }

      hid_layer[j].inneroutput[0] = sum1;        // output of first XOR unit
      hid_layer[j].inneroutput[1] = sum2;        // output of second XOR unit
      hid_layer[j].inneroutput[2] = THRESH;

      for (int k=0; k<HDN_UNTS+1; k++){
     // test if the product of the input and weight is positive or negative:
          if ( (hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k]) >=
0) {    // if weighted input is positive:
              if (hid_layer[j].innerweight[k] >= 1) {    // if weight is > 1:
                 sum += 1.0;                             // excitatory signal = 1
              }
              else {                                     // if weight is <= 1:
                 sum += -1.0;                            // inhibitory signal
              }
           }
           else {                          // if weighted input is negative:
                if ((abs (hid_layer[j].innerweight[k])) >= 1) {// |weight| > 1
                    sum += -1.0;                          // inhibitory signal
                }
                else {                         // if magnitude of weight is < 1
                    sum += 1.0;                           // excitatory signal
                }
           }
      }
      // ************************************************************
        hid_layer[j].output = sum;               // output of XOR unit
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
      fnet = -(sum)/NET_THRESHOLD;
     //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
      out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
```

```
     // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
     // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
     //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
     // out_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
     // sumg = ((sum)/3.0);                  // for gaussian/wavelet function
     // if (out_layer[j].output != out_layer[j].output) {      // nan error …
       //  out_layer[j].output = 1.0;
     // }
    }
  }
}
```

B.13 3-Shift Model

```
// ===========================================================================
// Description: This function propagates patterns through the Neural Network.
void propagate(int i)
{
  int c;
  float sum, sum1, sum2, fnet, fnet1, fnet2;
 // float tem, tem2, tem3;                              // for wavelet function
  // determine if # of input neurons is even or odd:
  if(IN_UNTS & 1)                                  // if odd # of input neurons:
    c = (IN_UNTS / 2) +1;
  else                                            // if even # of input neurons:
    c = IN_UNTS / 2;

  if(HDN_NU == 0)                   // if there are no hidden divcon neurons:
  {
    // ------- initialize and  calculate the output of the divcon neuron:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
      sum1=0.0;
      sum2=0.0;
      // ******* begin data processing inside each output divcon neuron:
      for(int k=0; k<c; k++)
        sum1 += out_layer[j].weight[k]*input[i][k];    // first inner neuron
      for(int k=c; k<IN_UNTS + 1; k++)
        sum2 += out_layer[j].weight[k]*input[i][k];    // second inner neuron

      fnet1 = (-sum1);
      fnet2 = (-sum2);

      out_layer[j].inneroutput[0] = 1.0/(1.0 + exp(fnet1)); // inner neuron 1
      out_layer[j].inneroutput[1] = 1.0/(1.0 + exp(fnet2)); // inner neuron 2
      out_layer[j].inneroutput[2] = THRESH;                 // bias

      for(int k=0; k<HDN_UNTS+1; k++)
        sum += out_layer[j].innerweight[k] * out_layer[j].inneroutput[k];
      // ***************************************************************
      fnet = -(sum);
```

```
    out_layer[j].output = 1.0/(1.0 + exp(fnet));    // output of DN
  }
}
else                                        // if hidden divcon neurons exist:
{
  // -------- initialize & calculate hidden divcon neurons' responses:
  for(int j=0; j<HDN_NU; j++)
  {
    sum=0.0;
    sum1=0.0;
    sum2=0.0;
    // **** begin data processing inside each hidden divcon neuron:******
    // half of the inputs are processed:
    for(int k=0; k<c; k++){
  // test if the product of the input and weight is positive or negative:
      if ( (hid_layer[j].weight[k]*input[i][k]) >= 0) { // weighted input +
         if (hid_layer[j].weight[k] > 1) {             // if weight is > 1:
            sum1 += input[i][k] * 2;                   // left shift
         }
         else {                                       // if weight is <= 1:
            if (hid_layer[j].weight[k] == 1.0) {      // if weight is = 1
                sum1 += input[i][k];                  // no change to input
            }
            else {                                    // if weight is < 1:
                sum1 += input[i][k] / 2;              // right shift
            }
         }
      }
      else {                             // if weighted input is negative:
         if ( (abs (hid_layer[j].weight[k])) > 1) {        // |weight| > 1
            sum1 += input[i][k] * 2;                       // left shift
         }
         else {                                           // |weight| <= 1:
            if ((abs (hid_layer[j].weight[k]))==1.0) {     // weight is = 1
               sum1 += input[i][k];                     // no change to input
            }
            else {                                      // if weight is < 1:
               sum1 += input[i][k] / 2;                 // right shift
            }
         }
      }
    }
    // the remaining half of the inputs are processed:
    for(int k=c; k< IN_UNTS + 1; k++) {
   // test if the product of the input and weight is positive or negative:
       if ((hid_layer[j].weight[k]*input[i][k]) >= 0) { // weighted input +
          if (hid_layer[j].weight[k] > 1) {             // if weight is > 1:
            sum2 += input[i][k] * 2;                    // left shift
          }
          else {                                        // if weight is <= 1:
           //  if (hid_layer[j].weight[k] == 1.0) {     // if weight is = 1:
                sum2 += input[i][k];                    // no change to input
             }
             else {                                     // if weight is < 1:
                sum2 += input[i][k] / 2;                // right shift
             }
          }
```

```
        }
        else {                              // if weighted input is negative:
          if ( (abs (hid_layer[j].weight[k])) > 1) {       // |weight| > 1:
             sum2 += input[i][k] * 2;                      // left shift
          }
          else {                                           // |weight| <= 1
             if ( (abs (hid_layer[j].weight[k])) == 1.0) {  // weight is = 1
                sum2 += input[i][k];                    // no change to input
             }
             else {                                       // if weight is < 1:
                sum2 += input[i][k] / 2;                // right shift
             }
          }
        }
      }

      hid_layer[j].inneroutput[0] = sum1;       // output of first shift unit
      hid_layer[j].inneroutput[1] = sum2;      // output of second shift unit
      hid_layer[j].inneroutput[2] = THRESH;

      for (int k=0; k<HDN_UNTS+1; k++){
     // test if the product of the input and weight is positive or negative:
         if ( (hid_layer[j].innerweight[k] * hid_layer[j].inneroutput[k]) >=
0) {    // if weighted input is positive:
             if (hid_layer[j].innerweight[k] > 1) {     // if weight is > 1:
                sum += hid_layer[j].inneroutput[k] * 2;     // left shift
             }
             else {                                    // if weight is <= 1:
                if (hid_layer[j].weight[k] == 1.0)   // if weight is = 1
                   sum += hid_layer[j].inneroutput[k]; // no change to input
                }
                else {                                   // if weight is < 1:
                   sum += hid_layer[j].inneroutput[k] / 2;    // right shift
                }
             }
          }
          else {                            // if weighted input is negative:
               if ( (abs (hid_layer[j].innerweight[k]))>1) { // |weight| > 1:
                  sum += hid_layer[j].inneroutput[k] * 2;    // left shift
               }
               else {                      // if magnitude of weight is < 1
                  if ((abs (hid_layer[j].innerweight[k]))==1.0) { // weight=1
                      sum+=hid_layer[j].inneroutput[k]; // no change to input
                  }
                  else {                                  // if weight is < 1
                      sum += hid_layer[j].inneroutput[k] / 2;  // right shift
                  }
               }
          }
      }
      // ************************************************************
        hid_layer[j].output = sum;                    // output of shift unit
    }
    // -------- initialize and calculate output neurons' responses:
    for(int j=0; j<OUT_NU; j++)
    {
      sum=0.0;
```

```
        for(int k=0; k<HDN_NU+1; k++)
          sum += out_layer[j].weight[k] * hid_layer[k].output;
       fnet = -(sum)/NET_THRESHOLD;
      //  out_layer[j].output = 1.0/(1.0 + exp(fnet));       // binary sigmoid
       out_layer[j].output = 2.0/(1.0 + exp(fnet)) - 1.0;   // bipolar sigmoid
      // out_layer[j].output = (exp(sum) - exp(fnet))/(exp(sum) + exp(fnet));
// hyperbolic tan
      // tem3 = ((sum)/3.0) * ((sum)/3.0);            // for wavelet function
      //  out_layer[j].output  =  cos(1.75  *  ((sum)/3.0))  *  exp(-tem3/2);
// wavelet function - b=0,a=3
      // out_layer[j].output = exp(-(sum * sum)/2);      // gaussian function
      // sumg = ((sum)/3.0);                 // for gaussian/wavelet function
      // if (out_layer[j].output != out_layer[j].output) {      // nan error …
        //  out_layer[j].output = 1.0;
      // }
     }
   }
}
```

Appendix C

HARDWARE DESIGN SOURCE CODE

C.1 Perceptron Model

```
----------------------------------------------------------------------------
-- Description:    crisp hidden neuron (perceptron)
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity crisp_hid_neuron is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (1 downto 0);
          write_addr: in std_logic_vector(1 downto 0);
          we: in  STD_LOGIC;
          clk: in  STD_LOGIC;
          rd: in std_logic;
          wt: in std_logic;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          result: out signed (7 downto 0);
          rst: in STD_LOGIC);
end crisp_hid_neuron;

architecture Behavioral of crisp_hid_neuron is

    component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
              p: out signed(15 downto 0)
       );
       end component;

       component ram_unit_block
       port ( clk: std_logic;
              en: std_logic;
              rd: std_logic;
              wt: std_logic;
              read_addr: in std_logic_vector(1 downto 0);
              write_addr: in std_logic_vector(1 downto 0);
              data_in: in std_logic_vector(7 downto 0);
              data_out: out std_logic_vector(7 downto 0)
       );
       end component;

       component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
              sum: out signed(15 downto 0)
       );
```

```
       end component;

      component reg16
       port (d: in signed(15 downto 0);
             q: out signed(15 downto 0);
             clk: in std_logic;
             en: in std_logic;
             rst: in std_logic
       );
      end component;

       component act_funct
       port ( result_in: in signed(15 downto 0);
              f_out: out signed(3 downto 0)
       );
       end component;

       signal w_carrier: std_logic_vector(7 downto 0);
       signal prod_carrier: signed (15 downto 0);
       signal sum_carrier: signed(15 downto 0);
       signal partial_sum: signed(15 downto 0);
       signal f_out: signed(3 downto 0);

begin
     adder: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);

        reg16_cmp: reg16 port map(d=>partial_sum, q=>sum_carrier, en=>en_acc,
clk=>slow_clk, rst=>rst);

        act_f: act_funct port map(result_in => sum_carrier, f_out => f_out);

        result <= "0000" & f_out;

        mul_comp:   mul8x8   port   map(x   =>   signed(input_i),   y   =>
signed(w_carrier), p => prod_carrier);

       M0:  ram_unit_block  port  map(data_in  =>  input_w,  read_addr  =>
read_addr, write_addr => write_addr, en => we, clk => clk, data_out =>
w_carrier, rd => rd, wt => wt);

end Behavioral;

----------------------------------------------------------------------------
-- Description: bipolar activation function with amplitude = 10 and shifted
--              up five units in order to output integers only.
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity act_funct is
    Port (result_in: in signed (15 downto 0);
          f_out: out signed (3 downto 0));
end act_funct;
```

```
architecture Behavioral of act_funct is
     signal p,n: std_logic;
        signal f: signed(3 downto 0);

begin
     p <= (not result_in(6) and not result_in(7) and not result_in(8) and not
result_in(9)   and   not   result_in(10)   and   not   result_in(11)   and   not
result_in(11)   and   not   result_in(12)   and   not   result_in(13)   and   not
result_in(14) and not result_in(15));
        n <= (result_in(6) and result_in(7) and result_in(8) and result_in(9)
and result_in(10) and result_in(11) and result_in(12) and result_in(13) and
result_in(14) and result_in(15));

        process(result_in, p,n)
        begin
          if(p='1' and result_in(5)='0') or (n='1' and result_in(5)='1') then
                 case result_in(5 downto 2) is
                         when "0000" => f<="0101";      -- when 0, 5
                               when "0001" => f<="0110";      -- when 1, 6
                               when "0010" => f<="0111";      -- when 2, 7
                               when "0011" => f<="1000";      -- when 3, 8
                               when "0100" => f<="1000";      -- when 4, 8
                                   when "0101" => f<="1001";  -- when 5, 9
                                   when "0110" => f<="1001";  -- when 6, 9
                                   when "0111" => f<="1010";  -- when 7, 10
                               when "1000" => f<="0000";      -- when -8, 0
                               when "1001" => f<="0000";      -- when -7, 0
                               when "1010" => f<="0000";      -- when -6, 0
                               when "1011" => f<="0000";      -- when -5, 0
                               when "1100" => f<="0001";      -- when -4, 1
                               when "1101" => f<="0010";   -- when -3, 2
                                  when "1110" => f<="0011";   -- when -2, 3
                                  when "1111" => f<="0101";   -- when -1, 5
                               when others => f<="0000";
                        end case;

                  else
                     if(result_in(15)='1') then
                           f<="0000";
                        else
                           f<="1010";
                        end if;
                  end if;
             end process;
             f_out<=f;
end Behavioral;
----------------------------------------------------------------------------
-- Description: 16-bit signed integer adder
----------------------------------------------------------------------------
library IEEE;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

entity add16 is
    Port (a: in signed (15 downto 0);
          b: in signed (15 downto 0);
```

```
          sum: out signed (15 downto 0));
end add16;

architecture Behavioral of add16 is
begin
    sum <= a + b;
end Behavioral;
----------------------------------------------------------------------------
-- Description:    16-bit register
----------------------------------------------------------------------------
library IEEE;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

entity reg16 is
    Port (d: in signed (15 downto 0);
          q: out signed (15 downto 0);
          clk: in STD_LOGIC;
          en: in STD_LOGIC;
          rst: in STD_LOGIC);
end reg16;

architecture Behavioral of reg16 is
     signal q_reg: signed (15 downto 0);
     signal q_next: signed (15 downto 0);
begin
    process (clk, rst)
       begin
             if (rst='1') then
                       q_reg <= "0000000000000000";
                elsif (clk'event and clk='1') then
                       q_reg <= q_next;
                   end if;
   end process;

   q_next <= d when en='1' else q_reg;
   q <= q_reg;

end Behavioral;
----------------------------------------------------------------------------
-- Description:  multiplier of two signed integers
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity mul8x8 is
    Port (x: in signed(7 downto 0);
          y: in signed(7 downto 0);
          p: out signed(15 downto 0));
end mul8x8;

architecture Behavioral of mul8x8 is
begin

  p <= x * y;
```

```
end Behavioral;
----------------------------------------------------------------------------
-- Description:    8-bitx4 RAM unit block -- single port ram
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ram_unit_block is
    generic( width: integer:=8;
             depth: integer:=4;
             addr:  integer:=2);

    Port (clk: in STD_LOGIC;
          en: in std_logic;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          read_addr: in std_logic_vector(addr-1 downto 0);
          write_addr: in std_logic_vector(addr-1 downto 0);
          data_in: in std_logic_vector(width-1 downto 0);
          data_out : out STD_LOGIC_vector(width-1 downto 0)
       );
end ram_unit_block;

architecture Behavioral of ram_unit_block is
     type ram_type is array (0 to depth-1) of
             std_logic_vector(width-1 downto 0);

     signal tmp_ram: ram_type;

begin
     -- read functional section:
     process(clk, rd)
         begin
            if (clk'event and clk='1') then
                if (en = '1') then
                    if (rd = '1') then
                        data_out <= tmp_ram(conv_integer(read_addr));
                    else
                        data_out <= (data_out'range => 'Z');
                    end if;
                end if;
            end if;
      end process;
    -- write functional section
      process(clk, wt)
      begin
           if(clk'event and clk='1') then
              if (en= '1') then
                  if (wt = '1') then
                      tmp_ram(conv_integer(write_addr)) <= data_in;
                  end if;
              end if;
           end if;
      end process;
```

```
end Behavioral;
----------------------------------------------------------------------------
-- Description:    crisp output neuron -- perceptron
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity crisp_out_neuron is
    Port (input_i: in signed (7 downto 0);
          input_w: in std_logic_vector (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (1 downto 0);
          write_addr: in std_logic_vector (1 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          rd: in std_logic;
          wt: in std_logic;
          result: out STD_LOGIC;
          rst: in STD_LOGIC);
end crisp_out_neuron;

architecture Behavioral of crisp_out_neuron is
     component mul8x8
        port ( x: in signed (7 downto 0);
               y: in signed (7 downto 0);
               p: out signed (15 downto 0)
        );
        end component;

     component ram_unit_block
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(1 downto 0);
                    write_addr: in std_logic_vector(1 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
     end component;

    component add16
        port (a: in signed (15 downto 0);
              b: in signed (15 downto 0);
              sum: out signed (15 downto 0)
        );
    end component;

    component reg16
        port (d: in signed (15 downto 0);
              q: out signed (15 downto 0);
              clk: in std_logic;
              en: in std_logic;
              rst: in std_logic
```

```
        );
    end component;

        signal weight_carrier: std_logic_vector (7 downto 0);
        signal prod_carrier: signed(15 downto 0);
        signal sum_carrier: signed(15 downto 0);
        signal partial_sum: signed(15 downto 0);

begin

adder: add16 port map(a=>prod_carrier, b=>sum_carrier, sum => partial_sum);
reg16_cmp: reg16 port map (d => partial_sum, q => sum_carrier, en => en_acc,
clk => slow_clk, rst => rst);

result <= (not sum_carrier(15)) and (sum_carrier(0) or sum_carrier(1) or
sum_carrier(2) or sum_carrier(3) or sum_carrier(4) or sum_carrier(5) or
sum_carrier(6) or sum_carrier(7) or sum_carrier(8));

mul_comp: mul8x8 port map (x => input_i, y => signed (weight_carrier), p =>
prod_carrier);

M1: ram_unit_block port map(data_in => input_w, read_addr => read_addr,
write_addr => write_addr, en => we, clk => clk, data_out => weight_carrier,
rd => rd, wt => wt);

end Behavioral;
```

C.2 Initial DN Model

```
----------------------------------------------------------------------------
-- Description:    Initial Divcon Neuron (DN) design using 3 perceptrons.
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity initial_DN is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (1 downto 0);
          write_addr: in STD_LOGIC_VECTOR (1 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
                    en1: in std_logic;
                    en2: in std_logic;
                    en3: in std_logic;
                    en4: in std_logic;
                    en5: in std_logic;
                    en6: in std_logic;
```

```
                    c_done:  in std_logic;
                    max_reached: in std_logic;
            rst: in  STD_LOGIC;
            result: out signed(7 downto 0));
end initial_DN;

architecture Behavioral of initial_DN is
    component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
              p: out signed(15 downto 0)
       );
    end component;

    component ram_unit_block
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(1 downto 0);
                    write_addr: in std_logic_vector(1 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
    end component;

    component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
              sum: out signed(15 downto 0)
       );
    end component;

    component reg16
       port (d: in signed(15 downto 0);
             q: out signed(15 downto 0);
                    clk: in std_logic;
                    en: in std_logic;
                    rst: in std_logic
       );
    end component;

    component act_funct
       port ( result_in: in signed(15 downto 0);
              f_out: out signed(3 downto 0)
       );
    end component;

    component decoder_1_to_2
       port ( I: in signed(15 downto 0);
              S: in std_logic;
                    O1: out signed(15 downto 0);
                    O2: out signed(15 downto 0)
       );
    end component;

    component demux8
```

```
      port ( I: in std_logic_vector(7 downto 0);
             S: in std_logic;
                   O1: out std_logic_vector(7 downto 0);
                   O2: out std_logic_vector(7 downto 0)
      );
    end component;

    component reg_8
      port ( d: in signed(7 downto 0);
             clk: in std_logic;
                   rst: in std_logic;
                   en: in std_logic;
                   q: out signed(7 downto 0)
      );
   end component;

   signal w_carrier: std_logic_vector(7 downto 0);              -- weights
   signal w_carrier7_ex: std_logic_vector(15 downto 0);         -- weights
   signal w_carrier4: std_logic_vector(7 downto 0);             -- weights
   signal w_carrier5: std_logic_vector(7 downto 0);             -- weights
   signal w_carrier6: std_logic_vector(7 downto 0);             -- weights
   signal w_carrier7: std_logic_vector(7 downto 0);             -- weights
   signal prod_carrier: signed (15 downto 0);             -- weighted input
   signal sum_carrier: signed(15 downto 0);               -- accumulated sum
   signal partial_sum: signed(15 downto 0);                   -- partial sum
   signal f_out3: signed(3 downto 0);                         -- DN output
   signal f_out1: signed(3 downto 0);       -- output of first half of inputs
   signal f_out1_ex: signed(7 downto 0);
   signal f_out2: signed(7 downto 0);                     -- output carrier
   signal prod_carrier1: signed(15 downto 0);             -- product carrier
   signal prod_carrier2: signed(15 downto 0);             -- product carrier A
   signal product_A: signed(15 downto 0);                 -- product A
   signal prod_carrier3: signed(15 downto 0);             -- product carrier B
   signal product_B: signed(15 downto 0);                 -- product B
   signal partial_sum1: signed(15 downto 0);              -- partial sum
   signal partial_sum2: signed(15 downto 0);              -- accumulated sum
   signal rst_carrier: std_logic;                         -- reset signal

begin

        Mem0: ram_unit_block port map(data_in => input_w, read_addr =>
read_addr, write_addr => write_addr, en => we, clk => clk, data_out =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier4, O2
=> w_carrier5);
        Mul0: mul8x8 port map(x => signed(input_i), y => signed(w_carrier4),
p => prod_carrier);
        Add1: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        act_f1: act_funct port map(result_in => sum_carrier, f_out =>
f_out1);
        f_out1_ex <= "0000" & f_out1;
        R1: reg_8 port map(d => f_out1_ex, q => f_out2, clk => slow_clk, en
=> en2, rst => rst);
```

```
        DEM1: demux8 port map(I => w_carrier5, S => en6, O1 => w_carrier6, O2
=> w_carrier7);
        Mul1: mul8x8 port map(x => f_out2, y => signed(w_carrier6), p =>
prod_carrier1);
        Deco2: decoder_1_to_2 port map(I => prod_carrier1, S => en3, O1 =>
prod_carrier2, O2 => prod_carrier3);
        R2: reg16 port map(d => prod_carrier2, q => product_A, clk =>
slow_clk, en => en4, rst => rst);
        R3: reg16 port map(d => prod_carrier3, q => product_B, clk =>
slow_clk, en => en5, rst => rst);
        Add2: add16 port map(a => product_B, b => product_A, sum =>
partial_sum1);
        w_carrier7_ex <= "00000000"  & w_carrier7;
        Add3: add16 port map(a => partial_sum1, b => signed(w_carrier7_ex),
sum => partial_sum2);
        act_f3: act_funct port map(result_in => partial_sum2, f_out =>
f_out3);
        -- output
        result <= "0000" & f_out3;

end Behavioral;
----------------------------------------------------------------------------
-- Description:    8-bit register
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity reg_8 is
    Port ( d : in  signed (7 downto 0);
           q : out  signed (7 downto 0);
           clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           en : in  STD_LOGIC);
end reg_8;

architecture Behavioral of reg_8 is
     signal q_reg: signed (7 downto 0);
     signal q_next: signed (7 downto 0);
begin
    process (clk, rst)
       begin
            if (rst='1') then
                     q_reg <= "00000000";
               elsif (clk'event and clk='1') then
                     q_reg <= q_next;
                  end if;
    end process;
    q_next <= d when en='1' else q_reg;
    q <= q_reg;
end Behavioral;
----------------------------------------------------------------------------
-- Description:    8-bit demultiplexer (1 to 2)
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity demux8 is
    Port (I: in STD_LOGIC_VECTOR (7 downto 0);
          S: in STD_LOGIC;
          O1: out STD_LOGIC_VECTOR (7 downto 0);
          O2: out STD_LOGIC_VECTOR (7 downto 0));
end demux8;

architecture Behavioral of demux8 is
begin
     process (S)
     begin
         case S is
                 when '0' => O1 <= I;
                 when '1' => O2 <= I;
                    when others => null;
            end case;
     end process;
end Behavioral;
----------------------------------------------------------------------------
-- Description:    16-bit 1 to 2 demultiplexer
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity decoder_1_to_2 is
    Port (I: in signed(15 downto 0);
          S: in std_logic;
          O1: out signed(15 downto 0);
          O2: out signed (15 downto 0));
end decoder_1_to_2;

architecture Behavioral of decoder_1_to_2 is
begin
     process (S)
     begin
         case S is
                 when '0' => O1 <= I;
                 when '1' => O2 <= I;
                    when others => null;
            end case;
     end process;
end Behavioral;
```

C.3 XOR Units Model

```
----------------------------------------------------------------------------
-- Description:    DN using XOR units
----------------------------------------------------------------------------
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity XOR_DN is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (1 downto 0);
          write_addr: in STD_LOGIC_VECTOR (1 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          en1: IN std_logic;
          en2: IN std_logic;
          en3: IN std_logic;
                    en4: IN std_logic;
                    en5: IN std_logic;
                    en6: IN std_logic;
                    c_done: in std_logic;
                    max_reached: in std_logic;
           rst: in  STD_LOGIC;
           result: out signed (7 downto 0));
end XOR_DN;

architecture Behavioral of XOR_DN is
    component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                    p: out signed(15 downto 0)
       );
    end component;

    component ram_unit_block
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(1 downto 0);
                    write_addr: in std_logic_vector(1 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
    end component;

    component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                    sum: out signed(15 downto 0)
       );
    end component;

    component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
```

```
                clk: in std_logic;
                en: in std_logic;
                rst: in std_logic
    );
  end component;

  component act_funct
    port ( result_in: in signed(15 downto 0);
           f_out: out signed(3 downto 0)
    );
  end component;

  component decoder_1_to_2
    port ( I: in signed(15 downto 0);
           S: in std_logic;
                O1: out signed(15 downto 0);
                O2: out signed(15 downto 0)
    );
  end component;

  component xor_unit
     port ( x: in signed(7 downto 0);
            w: in signed(7 downto 0);
                    xor_out: out signed (15 downto 0)
     );
  end component;

  component demux8
    port ( I: in std_logic_vector(7 downto 0);
           S: in std_logic;
                O1: out std_logic_vector(7 downto 0);
                O2: out std_logic_vector(7 downto 0)
    );
  end component;

 signal w_carrier: std_logic_vector(7 downto 0);                -- weights
 signal w_carrier1: std_logic_vector(7 downto 0);               -- weights
 signal w_carrier2: std_logic_vector(7 downto 0);               -- weights
 signal w_carrier3: std_logic_vector(7 downto 0);               -- weights
 signal w_carrier4: std_logic_vector(7 downto 0);               -- weights
 signal w_carrier4_ex: std_logic_vector(15 downto 0);           -- weights
 signal prod_carrier: signed (15 downto 0);              -- XOR unit output
 signal sum_carrier: signed(15 downto 0);                -- accumulated sum
 signal partial_sum: signed(15 downto 0);                -- partial sum
 signal f_out1_red: signed(7 downto 0);            -- accumulated sum 8-bit
 signal f_out3: signed(3 downto 0);                -- output of DN
 signal f_out1: signed(15 downto 0);              -- accumulated sum result
 signal prod_carrier1: signed(15 downto 0);            -- product carrier
 signal prod_carrier2: signed(15 downto 0);            -- product carrier 1
 signal prod_carrier3: signed(15 downto 0);            -- product carrier 2
 signal product_A: signed(15 downto 0);                -- product A
 signal product_B: signed(15 downto 0);                -- product B
 signal partial_sum1: signed(15 downto 0);             -- partial sum
 signal partial_sum2: signed(15 downto 0);
 signal rst_carrier: std_logic;                     -- reset carrier signal

begin
```

```
     Mem0:  ram_unit_block  port  map(data_in  =>  input_w,  read_addr  =>
read_addr,  write_addr  =>  write_addr,  en  =>  we,  clk  =>  clk,  data_out  =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        XOR_U0:   xor_unit   port   map(x   =>   signed(input_i),   w   =>
signed(w_carrier1), xor_out => prod_carrier);
        Add1: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        R1: reg16 port map(d => sum_carrier, q => f_out1, clk => slow_clk, en
=> en2, rst => rst);
        f_out1_red <= f_out1(7 downto 0);
     DEM1: demux8 port map(I => w_carrier2, S => en3, O1 => w_carrier3, O2 =>
w_carrier4);
        Mul1: mul8x8 port map(x => f_out1_red, y => signed(w_carrier3), p =>
prod_carrier1);
        Deco0: decoder_1_to_2 port map(I => prod_carrier1, S => en4, O1 =>
prod_carrier2, O2 => prod_carrier3);
        R2: reg16 port map(d => prod_carrier2, q => product_A, clk =>
slow_clk, en => en5, rst => rst);
        R3: reg16 port map(d => prod_carrier3, q => product_B, clk =>
slow_clk, en => en6, rst => rst);
        Add2: add16 port map(a => product_B, b => product_A, sum =>
partial_sum1);
        w_carrier4_ex <= "00000000"  & w_carrier4;
        Add3: add16 port map(a => signed(w_carrier4_ex), b => partial_sum1,
sum => partial_sum2);
        act_f3: act_funct port map(result_in => partial_sum2, f_out =>
f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
----------------------------------------------------------------------------
-- Description:    Xor unit
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity xor_unit is
    Port ( x : in  signed (7 downto 0);
           w : in  signed (7 downto 0);
           xor_out : out  signed (15 downto 0));
end xor_unit;

architecture Behavioral of xor_unit is
begin
     process (x,w)
        begin
          if (x(7) = '0') then                  -- if input is positive:
             if (w(7) = '0') then   -- and w is +, product is positive (-1):
                  if (w >= 1) then                 -- if weight is > 1 (1):
                     xor_out <= "0000000000000001";   -- (excitatory signal)
```

```
                else                                -- if weight is < 1 (-1)
                   xor_out <= "1111111111111111";  --  (inhibitory signal)
                end if;

            else      -- if weight is negative, then product is negative (1):
                if (abs(w) >= "00000001") then    -- if |weight| is > 1(1):
                   xor_out <= "1111111111111111";    -- (inhibitory signal)
                else                              -- if |weight| is < 1 (-1):
                   xor_out <= "0000000000000001";    -- (excitatory signal)
                end if;
            end if;

         else                                     -- if input is negative:
            if (w(7) = '0') then      -- and w is +, product is negative (1):
                if (w >= "00000001") then         -- if weight is > 1 (1):
                   xor_out <= "1111111111111111";   -- (inhibitory signal)
                else                              -- if weight is < 1 (-1):
                   xor_out <= "0000000000000001";   -- (excitatory signal)
                end if;
            else     -- if weight is negative, then product is positive (-1):
                if (abs(w) >= "00000001") then   -- if |weight| is > 1 (1):
                   xor_out <= "0000000000000001";    -- (excitatory signal)
                else                              -- if |weight| is < 1 (-1):
                   xor_out <= "1111111111111111";    -- (inhibitory signal)
                end if;
            end if;
         end if;
    end process;
end Behavioral;
```

C.4 Shift Units Model

```
----------------------------------------------------------------------------
-- Description:    shift unit
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity shift_DN is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (1 downto 0);
          write_addr: in STD_LOGIC_VECTOR (1 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          en1: in STD_LOGIC;
          en2: in STD_LOGIC;
          en3: in STD_LOGIC;
```

```
                    en4: in STD_LOGIC;
                    en5: in STD_LOGIC;
                    en6: in STD_LOGIC;
            c_done: in STD_LOGIC;
            max_reached: in STD_LOGIC;
                    rst: in STD_LOGIC;
            result: out signed (7 downto 0));
end shift_DN;

architecture Behavioral of shift_DN is
    component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
              p: out signed(15 downto 0)
       );
    end component;

    component ram_unit_block
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(1 downto 0);
                    write_addr: in std_logic_vector(1 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
    end component;

    component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                    sum: out signed(15 downto 0)
       );
    end component;

    component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                    clk: in std_logic;
                    en: in std_logic;
                    rst: in std_logic
       );
    end component;

    component act_funct
       port ( result_in: in signed(15 downto 0);
              f_out: out signed(3 downto 0)
       );
    end component;

    component decoder_1_to_2
       port ( I: in signed(15 downto 0);
              S: in std_logic;
                    O1: out signed(15 downto 0);
                    O2: out signed(15 downto 0)
       );
```

```
    end component;

    component shitf_unit
        port ( x: in signed(7 downto 0);
               w: in signed(7 downto 0);
                        shift_out: out signed (15 downto 0)
        );
    end component;

    component demux8
        port ( I: in std_logic_vector(7 downto 0);
               S: in std_logic;
                     O1: out std_logic_vector(7 downto 0);
                     O2: out std_logic_vector(7 downto 0)
        );
    end component;

    signal w_carrier: std_logic_vector(7 downto 0);               -- weights
    signal w_carrier1: std_logic_vector(7 downto 0);              -- weights
    signal w_carrier2: std_logic_vector(7 downto 0);              -- weights
    signal w_carrier3: std_logic_vector(7 downto 0);              -- weights
    signal w_carrier4: std_logic_vector(7 downto 0);              -- weights
    signal w_carrier4_ex: std_logic_vector(15 downto 0);
    signal prod_carrier: signed (15 downto 0);           -- shift unit output
    signal sum_carrier: signed(15 downto 0);             -- accumulated sum
    signal partial_sum: signed(15 downto 0);             -- partial sum
    signal f_out1: signed(15 downto 0);                  -- partial output
    signal f_out1_red: signed(7 downto 0);
    signal f_out3: signed(3 downto 0);
    signal prod_carrier1: signed(15 downto 0);           -- product carrier
    signal prod_carrier2: signed(15 downto 0);           -- product carrier 2
    signal prod_carrier3: signed(15 downto 0);           -- product carrier 3
    signal product_A: signed(15 downto 0);               -- product A
    signal product_B: signed(15 downto 0);               -- product B
    signal partial_sum1: signed(15 downto 0);            -- partial sum
    signal partial_sum2: signed(15 downto 0);            -- partial sum 2
    signal rst_carrier: std_logic;                     -- reset carrier signal

begin

     Mem0:  ram_unit_block  port  map(data_in  =>  input_w,  read_addr  =>
read_addr,  write_addr  =>  write_addr,  en  =>  we,  clk  =>  clk,  data_out  =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        SHIFT_U0:   shitf_unit   port   map(x   =>   signed(input_i),   w   =>
signed(w_carrier1), shift_out => prod_carrier);
        Add1: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0:  reg16  port  map(d  =>  partial_sum,  q  =>  sum_carrier,  clk  =>
slow_clk, en => en_acc, rst => rst_carrier);
        R1: reg16 port map(d => sum_carrier, q => f_out1, clk => slow_clk, en
=> en2, rst => rst);
        f_out1_red <= f_out1(7 downto 0);
        DEM1: demux8 port map(I => w_carrier2, S => en3, O1 => w_carrier3, O2
=> w_carrier4);
```

```
        Mul1: mul8x8 port map(x => f_out1_red, y => signed(w_carrier3), p =>
prod_carrier1);
        Deco1: decoder_1_to_2 port map(I => prod_carrier1, S => en4, O1 =>
prod_carrier2, O2 => prod_carrier3);
        R2: reg16 port map(d => prod_carrier2, q => product_A, clk =>
slow_clk, en => en5, rst => rst);
     R3: reg16 port map(d => prod_carrier3, q => product_B, clk => slow_clk,
en => en6, rst => rst);
        Add2: add16 port map(a => product_B, b => product_A, sum =>
partial_sum1);
        w_carrier4_ex <= "00000000"  & w_carrier4;
        Add3: add16 port map(a => signed(w_carrier4_ex), b => partial_sum1,
sum => partial_sum2);
        act_f3: act_funct port map(result_in => partial_sum2, f_out =>
f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
----------------------------------------------------------------------------
-- Description:    shift unit
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity shitf_unit is
    Port (x: in signed (7 downto 0);
          w: in signed (7 downto 0);
          shift_out: out signed (15 downto 0));
end shitf_unit;

architecture Behavioral of shitf_unit is
begin

     process (x,w)
        begin
             if (x(7) = '0') then                    -- if input is positive:
               if (w(7) = '0') then       -- and w is +, product is positive:
                  if (w > 1) then                   -- if weight is > 1:
                     shift_out <= x * 2;            -- left shift
                  else                              -- if weight is <= 1:
                     if ( w = 1) then              -- if weight = 1:
                        shift_out <= "00000000" & x;    -- no change to input
                     else                           -- if weight is < 1:
                        shift_out <= x / 2;        -- right shift
                     end if;
                  end if;
              else                     -- if w is -, then product is negative:
                  if (abs(w) > 1) then           -- if |weight| is > 1:
                        shift_out <= x * 2;          -- left shift
                  else                               -- if |weight| is <= 1:
                     if (abs(w) = 1) then           -- if |weight| is = 1:
                        shift_out <= "00000000" & x;    -- no change to input
                     else                           -- if |weight| is < 1:
                        shift_out <= x / 2;         -- right shift
                     end if;
```

```
                end if;
            end if;
          else                                    -- if input is negative:
            if (w(7) = '0') then          -- and w is +, product is negative:
                if (w > 1) then                    -- if weight is > 1:
                   shift_out <= x * 2;             -- left shift
                else                               -- if weight is <= 1:
                   if ( w = 1) then                -- if weight is = 1:
                      shift_out <= "11111111" & x;    -- no change to input
                   else                            -- if weight is < 1:
                      shift_out <= x / 2;          -- right shift
                   end if;
                end if;
            else                                  -- if w is +, product is +:
                if (abs(w) > 1 ) then                -- if |weight| is > 1:
                      shift_out <= x * 2;            -- left shift
                else                                 -- if |weight| is <= 1:
                   if ( w = 1) then                  -- if |weight| is = 1:
                      shift_out <= "11111111" & x;    -- no change to input
                   else                              -- if weight is < 1:
                      shift_out <= x / 2;            -- right shift
                   end if;
                end if;

            end if;
          end if;
        end process;
end Behavioral;
```

C.5 Quantum Neurons Model

```
----------------------------------------------------------------------------
-- Description:    quantum neuron.
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity quantum_DN is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (2 downto 0);
          write_addr: in STD_LOGIC_VECTOR (2 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          rst: in STD_LOGIC;
          en1: in STD_LOGIC;
          en2: in STD_LOGIC;
          en3: in STD_LOGIC;
```

```
                    en4: in std_logic;
                    en5: in std_logic;
                    en6: in std_logic;
            c_done: in STD_LOGIC;
            result: out signed (7 downto 0));
end quantum_DN;

architecture Behavioral of quantum_DN is
    component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                    p: out signed(15 downto 0)
       );
    end component;

    component ram_unit_block_2
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(2 downto 0);
                    write_addr: in std_logic_vector(2 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
    end component;

    component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                    sum: out signed(15 downto 0)
       );
    end component;

    component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                    clk: in std_logic;
                    en: in std_logic;
                    rst: in std_logic
       );
    end component;

    component act_funct
       port ( result_in: in signed(15 downto 0);
              f_out: out signed(3 downto 0)
       );
    end component;

    component decoder_1_to_2
       port ( I: in signed(15 downto 0);
              S: in std_logic;
                    O1: out signed(15 downto 0);
                    O2: out signed(15 downto 0)
       );
    end component;
```

```
    component demux8
        port ( I: in std_logic_vector(7 downto 0);
               S: in std_logic;
                    O1: out std_logic_vector(7 downto 0);
                    O2: out std_logic_vector(7 downto 0)
        );
    end component;

    signal w_carrier: std_logic_vector(7 downto 0);                -- weights
    signal w_carrier1: std_logic_vector(7 downto 0);               -- weights
    signal w_carrier2: std_logic_vector(7 downto 0);               -- weights
    signal w_carrier3: std_logic_vector(7 downto 0);               -- weights
    signal w_carrier4: std_logic_vector(7 downto 0);               -- weights
    signal w_carrier4_ex: std_logic_vector(15 downto 0);
    signal prod_carrier: signed (15 downto 0);                -- weighted input
    signal sum_carrier: signed(15 downto 0);                 -- accumulated sum
    signal partial_sum: signed(15 downto 0);                 -- sum
    signal hid_sum1: signed(15 downto 0);                   -- accumulated sum 1
    signal carrier_hid_sum1: signed(15 downto 0);           -- sum 1 carrier
    signal hid_sum2: signed(15 downto 0);                   -- accumulated sum 2
    signal carrier_hid_sum2: signed(15 downto 0);           -- sum 2 carrier
    signal f_out3: signed(3 downto 0);                             -- output
    signal prod_carrier1: signed(15 downto 0);              -- weighted input
    signal partial_sum1: signed(15 downto 0);               -- hidden output
    signal partial_sum1_red: signed(7 downto 0);
    signal partial_sum3: signed(15 downto 0);                   -- partial sum
    signal partial_sum4: signed(15 downto 0);                   -- partial sum
    signal result_A: signed(15 downto 0);                       -- result A
    signal carrier_resultA: signed(15 downto 0);            -- result A carrier
    signal result_B: signed(15 downto 0);                   -- result B
    signal carrier_resultB: signed(15 downto 0);            -- result B carrier
    signal rst_carrier: std_logic;                          -- reset signal

begin

        Mem0: ram_unit_block_2 port map(data_in => input_w, read_addr =>
read_addr, write_addr => write_addr, en => we, clk => clk, data_out =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        Mul0: mul8x8 port map(x => signed(input_i), y => signed(w_carrier1),
p => prod_carrier);
        Add1: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        Deco1: decoder_1_to_2 port map(I => sum_carrier, S => en2, O1 =>
hid_sum1, O2 => hid_sum2);
        R4: reg16 port map(d => hid_sum1, q => carrier_hid_sum1, clk =>
slow_clk, en => en5, rst => rst);
        R5: reg16 port map(d => hid_sum2, q => carrier_hid_sum2, clk =>
slow_clk, en => en6, rst => rst);
        Add2: add16 port map(a => carrier_hid_sum2, b => carrier_hid_sum1,
sum => partial_sum1);
        DEM01: demux8 port map(I => w_carrier2, S => en3, O1 => w_carrier3,
O2 => w_carrier4);
```

```
        partial_sum1_red <= partial_sum1 (7 downto 0);
        Mul1: mul8x8 port map(x => signed(w_carrier3), y => partial_sum1_red,
p => prod_carrier1);
        Deco2: decoder_1_to_2 port map(I => prod_carrier1, S => en4, O1 =>
result_A, O2 => result_B);
        R2: reg16 port map(d => result_A, q => carrier_resultA, clk =>
slow_clk, en => en_acc, rst => rst);
        R3: reg16 port map(d => result_B, q => carrier_resultB, clk =>
slow_clk, en => en_acc, rst => rst);
        Add3: add16 port map(a => carrier_resultB, b => carrier_resultA, sum
=> partial_sum3);
        w_carrier4_ex <= "00000000"  & w_carrier4;
        Add4: add16 port map(a => signed(w_carrier4_ex), b => partial_sum3,
sum => partial_sum4);
        act_f3: act_funct port map(result_in => partial_sum4, f_out =>
f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
----------------------------------------------------------------------------
-- Description:    8-bitx8 RAM unit block -- single port ram
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ram_unit_block_2 is
    generic( width: integer:=8;
                depth: integer:=8;
                         addr:  integer:=3);

    Port (clk: in STD_LOGIC;
          en: in std_logic;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
                    read_addr: in std_logic_vector(addr-1 downto 0);
                    write_addr: in std_logic_vector(addr-1 downto 0);
                    data_in: in std_logic_vector(width-1 downto 0);
           data_out: out STD_LOGIC_vector(width-1 downto 0)
       );

end ram_unit_block_2;

architecture Behavioral of ram_unit_block_2 is
     type ram_type is array (0 to depth-1) of
             std_logic_vector(width-1 downto 0);

     signal tmp_ram: ram_type;

begin
     -- read functional section:
     process(clk, rd)
         begin
            if (clk'event and clk='1') then
               if (en = '1') then
                  if (rd = '1') then
```

```
                    data_out <= tmp_ram(conv_integer(read_addr));
                 else
                    data_out <= (data_out'range => 'Z');
                 end if;
              end if;
           end if;
      end process;
           -- write functional section
      process(clk, wt)
      begin
           if(clk'event and clk='1') then
              if (en= '1') then
                  if (wt = '1') then
                      tmp_ram(conv_integer(write_addr)) <= data_in;
                  end if;
              end if;
           end if;
      end process;
end Behavioral;
```

C.6 XOR & Shift Model

```
----------------------------------------------------------------------------
-- Description:   Divcon neuron model using 1 XOR unit and 1 shift unit
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity xor_and_shift is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (1 downto 0);
          write_addr: in STD_LOGIC_VECTOR (1 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          rst: in STD_LOGIC;
          en1: in STD_LOGIC;
          en2: in STD_LOGIC;
          en3: in STD_LOGIC;
          en4: in STD_LOGIC;
          en5: in STD_LOGIC;
          en6: in STD_LOGIC;
          en7: in std_logic;
          c_done: in std_logic;
          result: out  signed (7 downto 0));
end xor_and_shift;

architecture Behavioral of xor_and_shift is
```

```
component mul8x8
    port ( x: in signed(7 downto 0);
           y: in signed(7 downto 0);
                 p: out signed(15 downto 0)
    );
end component;

component ram_unit_block
    port ( clk: std_logic;
           en: std_logic;
                 rd: std_logic;
                 wt: std_logic;
                 read_addr: in std_logic_vector(1 downto 0);
                 write_addr: in std_logic_vector(1 downto 0);
                 data_in: in std_logic_vector(7 downto 0);
                 data_out: out std_logic_vector(7 downto 0)
    );
end component;

component add16
    port ( a: in signed(15 downto 0);
           b: in signed(15 downto 0);
                 sum: out signed(15 downto 0)
    );
end component;

component reg16
    port (d: in signed(15 downto 0);
           q: out signed(15 downto 0);
                 clk: in std_logic;
                 en: in std_logic;
                 rst: in std_logic
    );
end component;

component act_funct
    port ( result_in: in signed(15 downto 0);
           f_out: out signed(3 downto 0)
    );
end component;

component decoder_1_to_2
    port ( I: in signed(15 downto 0);
           S: in std_logic;
                 O1: out signed(15 downto 0);
                 O2: out signed(15 downto 0)
    );
end component;

component demux8
    port ( I: in std_logic_vector(7 downto 0);
           S: in std_logic;
                 O1: out std_logic_vector(7 downto 0);
                 O2: out std_logic_vector(7 downto 0)
    );
end component;
```

```
	component xor_unit
	  port ( x: in signed(7 downto 0);
	         w: in signed(7 downto 0);
	                  xor_out: out signed (15 downto 0)
	  );
	end component;


	component shitf_unit
	  port ( x: in signed(7 downto 0);
	         w: in signed(7 downto 0);
	                  shift_out: out signed (15 downto 0)
	  );
	end component;

	component mux2x1
	  port ( I0: in signed(15 downto 0);
	         I1: in signed(15 downto 0);
	                  S0: in std_logic;
	                  Y0: out signed(15 downto 0)
	  );
	end component;

	signal w_carrier: std_logic_vector(7 downto 0);             -- weights
	signal w_carrier1: std_logic_vector(7 downto 0);            -- weights
	signal w_carrier2: std_logic_vector(7 downto 0);            -- weights
	signal w_carrier3: std_logic_vector(7 downto 0);            -- weights
	signal w_carrier4: std_logic_vector(7 downto 0);            -- weights
	signal w_carrier5: std_logic_vector(7 downto 0);            -- weights
	signal w_carrier6: std_logic_vector(7 downto 0);            -- weights
	signal w_carrier6_ex: std_logic_vector(15 downto 0);        -- weights
	signal input_carrier1: std_logic_vector(7 downto 0);--input to XOR unit
	signal input_carrier2: std_logic_vector(7 downto 0);--input to SH unit
	signal prod_carrier: signed (15 downto 0);        -- output of xor unit
	signal sum_carrier: signed(15 downto 0);          -- accumulated sum
	signal partial_sum: signed(15 downto 0);          -- sum
	signal f_out1: signed(15 downto 0);               -- accumulated sum
	signal f_out1_red: signed(7 downto 0);
	signal f_out3: signed(3 downto 0);                -- output
	signal prod_carrier1: signed(15 downto 0);      -- output of shift unit
	signal prod_carrier2: signed(15 downto 0);      -- weighted input
	signal prod_carrier3: signed(15 downto 0);      -- weighted input
	signal partial_sum1: signed(15 downto 0);       -- hidden output
	signal partial_sum2: signed(15 downto 0);       -- partial sum
	signal prod_A: signed(15 downto 0);             -- result A
	signal carrier_resultA: signed(15 downto 0);    -- result A carrier
	signal prod_B: signed(15 downto 0);             -- result B
	signal carrier_resultB: signed(15 downto 0);    -- result B carrier
	signal rst_carrier: std_logic;                  -- reset signal

begin

     Mem0:  ram_unit_block  port  map(data_in  =>  input_w,  read_addr  =>
read_addr,  write_addr  =>  write_addr,  en  =>  we,  clk  =>  clk,  data_out  =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
```

```
        DEM1: demux8 port map(I => w_carrier1, S => en2, O1 => w_carrier3, O2
=> w_carrier4);
        DEM2: demux8 port map(I => input_i, S => en2, O1 => input_carrier1,
O2 => input_carrier2);
        XOR_0:  xor_unit  port  map(x  =>  signed(input_carrier1),  w  =>
signed(w_carrier3), xor_out => prod_carrier);
        SHIFT_0:  shitf_unit  port  map(x  =>  signed(input_carrier2),  w  =>
signed(w_carrier4), shift_out => prod_carrier1);
        MUX_0: mux2x1 port map(I0 => prod_carrier, I1 => prod_carrier1, S0 =>
en3, Y0 => prod_carrier2);
        Add0: add16 port map(a => prod_carrier2, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0:  reg16  port  map(d  =>  partial_sum,  q  =>  sum_carrier,  clk  =>
slow_clk, en => en_acc, rst => rst_carrier);
        R1: reg16 port map(d => sum_carrier, q => f_out1, clk => slow_clk, en
=> en4, rst => rst);
        DEM3: demux8 port map(I => w_carrier2, S => en5, O1 => w_carrier5, O2
=> w_carrier6);
        f_out1_red <= f_out1(7 downto 0);
        Mul0: mul8x8 port map(x => f_out1_red, y => signed(w_carrier5), p =>
prod_carrier3);
        Deco1: decoder_1_to_2 port map(I => prod_carrier3, S => en6, O1 =>
prod_A, O2 => prod_B);
        R2:  reg16  port  map(d  =>  prod_A,  q  =>  carrier_resultA,  clk  =>
slow_clk, en => en6, rst => rst);
        R3:  reg16  port  map(d  =>  prod_B,  q  =>  carrier_resultB,  clk  =>
slow_clk, en => en7, rst => rst);
        Add2: add16 port map(a => carrier_resultA, b => carrier_resultB, sum
=> partial_sum1);
        w_carrier6_ex <= "00000000" & w_carrier6;
        Add3: add16 port map(a => signed(w_carrier6_ex), b => partial_sum1,
sum => partial_sum2);
        act_f3:  act_funct  port  map(result_in  =>  partial_sum2,  f_out  =>
f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
----------------------------------------------------------------------------
-- Description:    16-bit 2 to 1 Mux
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity mux2x1 is
    Port ( I0 : in  Signed (15 downto 0);
           I1 : in  Signed (15 downto 0);
           Y0 : out  signed (15 downto 0);
           S0 : in  STD_LOGIC);
end mux2x1;

architecture Behavioral of mux2x1 is
begin

     process (S0)
```

```
     begin
         case S0 is
                 when '0' => Y0 <= I0;
                 when '1' => Y0 <= I1;
                    when others => null;
           end case;
     end process;
end Behavioral;
```

C.7 XOR & Quantum Model

```
----------------------------------------------------------------------------
-- Description: DN design containing 1 XOR unit and 1 Quantum neuron.
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity XOR_and_QU_DN is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (2 downto 0);
          write_addr: in STD_LOGIC_VECTOR (2 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          rst: in STD_LOGIC;
          en1: in STD_LOGIC;
          en2: in STD_LOGIC;
          en3: in STD_LOGIC;
          en4: in STD_LOGIC;
          en5: in STD_LOGIC;
          en6: in STD_LOGIC;
                    en7: in STD_LOGIC;
                    en8: in STD_LOGIC;
                    en9: in STD_LOGIC;
                    en10: in STD_LOGIC;
                    en11: in STD_LOGIC;
                    en12: in STD_LOGIC;
           c_done: in STD_LOGIC;
           result: out Signed (7 downto 0));
end XOR_and_QU_DN;

architecture Behavioral of XOR_and_QU_DN is
   component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                    p: out signed(15 downto 0)
       );
   end component;
```

```
component ram_unit_block_2
    port ( clk: std_logic;
           en: std_logic;
                 rd: std_logic;
                 wt: std_logic;
                 read_addr: in std_logic_vector(2 downto 0);
                 write_addr: in std_logic_vector(2 downto 0);
                 data_in: in std_logic_vector(7 downto 0);
                 data_out: out std_logic_vector(7 downto 0)
    );
end component;

component add16
    port ( a: in signed(15 downto 0);
           b: in signed(15 downto 0);
                 sum: out signed(15 downto 0)
    );
end component;

component reg16
    port (d: in signed(15 downto 0);
           q: out signed(15 downto 0);
                 clk: in std_logic;
                 en: in std_logic;
                 rst: in std_logic
    );
end component;

component act_funct
    port ( result_in: in signed(15 downto 0);
           f_out: out signed(3 downto 0)
    );
end component;

component decoder_1_to_2
    port ( I: in signed(15 downto 0);
           S: in std_logic;
                 O1: out signed(15 downto 0);
                 O2: out signed(15 downto 0)
    );
end component;

component demux8
    port ( I: in std_logic_vector(7 downto 0);
           S: in std_logic;
                 O1: out std_logic_vector(7 downto 0);
                 O2: out std_logic_vector(7 downto 0)
    );
end component;

component xor_unit
     port ( x: in signed(7 downto 0);
            w: in signed(7 downto 0);
                    xor_out: out signed (15 downto 0)
     );
end component;
```

```
   component mux2x1
        port ( I0: in signed(15 downto 0);
               I1: in signed(15 downto 0);
                        S0: in std_logic;
                        Y0: out signed(15 downto 0)
        );
   end component;

   signal w_carrier: std_logic_vector(7 downto 0);                -- weights
   signal w_carrier1: std_logic_vector(7 downto 0);               -- weights
   signal w_carrier2: std_logic_vector(7 downto 0);               -- weights
   signal w_carrier3: std_logic_vector(7 downto 0);               -- weights
   signal w_carrier4: std_logic_vector(7 downto 0);               -- weights
   signal w_carrier5: std_logic_vector(7 downto 0);               -- weights
   signal w_carrier6: std_logic_vector(7 downto 0);               -- weights
   signal w_carrier6_ex: std_logic_vector(15 downto 0);           -- weights
   signal input_carrier1: std_logic_vector(7 downto 0); -- input to XOR unit
   signal input_carrier2: std_logic_vector(7 downto 0);-- input to shift unit
   signal prod_carrier: signed (15 downto 0);            -- output of xor unit
   signal sum_carrier: signed(15 downto 0);              -- accumulated sum
   signal partial_sum: signed(15 downto 0);                                -- sum
   signal f_out1: signed(15 downto 0);                   -- accumulated sum
   signal f_out2: signed(15 downto 0);                   -- accumulated sum
   signal prod_carrier5_red: signed(7 downto 0);
   signal f_out3: signed(3 downto 0);                                      -- output
   signal prod_carrier1: signed(15 downto 0);         -- output of shift unit
   signal prod_carrier2: signed(15 downto 0);         -- weighted input
   signal prod_carrier3: signed(15 downto 0);         -- weighted input
   signal prod_carrier4: signed(15 downto 0);         -- weighted input
   signal prod_carrier5: signed(15 downto 0);         -- product carrier
   signal prod_carrier6: signed(15 downto 0);         -- product carrier
   signal sumA: signed(15 downto 0);                  -- accumulated sum A
   signal sumB: signed(15 downto 0);                  -- accumulated sum B
   signal partial_sum1: signed(15 downto 0);          -- hidden quantum output
   signal partial_sum2: signed(15 downto 0);          -- partial sum
   signal partial_sum3: signed(15 downto 0);          -- partial sum
   signal prod_A: signed(15 downto 0);                             -- result A
   signal carrier_resultA: signed(15 downto 0);          -- result A carrier
   signal prod_B: signed(15 downto 0);                   -- result B
   signal carrier_resultB: signed(15 downto 0);          -- result B carrier
   signal rst_carrier: std_logic;                        -- reset signal

begin

     Mem0: ram_unit_block_2 port map(data_in => input_w, read_addr =>
read_addr, write_addr => write_addr, en => we, clk => clk, data_out =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        DEM1: demux8 port map(I => w_carrier1, S => en2, O1 => w_carrier3, O2
=> w_carrier4);
        DEM2: demux8 port map(I => input_i, S => en2, O1 => input_carrier1,
O2 => input_carrier2);
        XOR_0: xor_unit port map(x => signed(input_carrier1), w =>
signed(w_carrier3), xor_out => prod_carrier);
        Mul0: mul8x8 port map(x => signed(input_carrier2), y =>
signed(w_carrier4), p => prod_carrier1);
```

```
        MUX_0: mux2x1 port map(I0 => prod_carrier, I1 => prod_carrier1, S0 =>
en3, Y0 => prod_carrier2);
        Add0: add16 port map(a => prod_carrier2, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        Deco0: decoder_1_to_2 port map(I => sum_carrier, S => en4, O1 =>
prod_carrier3, O2 => prod_carrier4);
        Deco1: decoder_1_to_2 port map(I => prod_carrier3, S => en5, O1 =>
sumA, O2 => sumB);
        R1: reg16 port map(d => sumA, q => f_out1, clk => slow_clk, en =>
en6, rst => rst);
        R2: reg16 port map(d => sumB, q => f_out2, clk => slow_clk, en =>
en7, rst => rst);
        Add1: add16 port map(a => f_out1, b => f_out2, sum => partial_sum1);
        MUX_1: mux2x1 port map(I0 => partial_sum1, I1 => prod_carrier4, S0 =>
en8, Y0 => prod_carrier5);
        prod_carrier5_red <= prod_carrier5(7 downto 0);
        DEM3: demux8 port map(I => w_carrier2, S => en9, O1 => w_carrier5, O2
=> w_carrier6);
        Mul1:   mul8x8   port   map(x   =>   prod_carrier5_red,   y   =>
signed(w_carrier5), p => prod_carrier6);
        Deco2: decoder_1_to_2 port map(I => prod_carrier6, S => en10, O1 =>
prod_A, O2 => prod_B);
        R3: reg16 port map(d => prod_A, q => carrier_resultA, clk =>
slow_clk, en => en11, rst => rst);
        R4: reg16 port map(d => prod_B, q => carrier_resultB, clk =>
slow_clk, en => en12, rst => rst);
        Add2: add16 port map(a => carrier_resultA, b => carrier_resultB, sum
=> partial_sum2);
        w_carrier6_ex <= "00000000" & w_carrier6;
        Add3: add16 port map(a => signed(w_carrier6_ex), b => partial_sum2,
sum => partial_sum3);
        act_f3: act_funct port map(result_in => partial_sum3, f_out =>
f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
```

C.8 Shift & Quantum Model

```
----------------------------------------------------------------------------
-- Description:    1 shift unit, and 1 quantum neuron
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity shift_and_QU_DN is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (2 downto 0);
```

```
           write_addr: in STD_LOGIC_VECTOR (2 downto 0);
           we: in STD_LOGIC;
           clk: in STD_LOGIC;
           rd: in STD_LOGIC;
           wt: in STD_LOGIC;
           slow_clk: in STD_LOGIC;
           en_acc: in STD_LOGIC;
           c_done: in std_logic;
            rst: in STD_LOGIC;
            en1: in STD_LOGIC;
            en2: in STD_LOGIC;
            en3: in STD_LOGIC;
            en4: in STD_LOGIC;
            en5: in STD_LOGIC;
            en6: in STD_LOGIC;
            en7: in STD_LOGIC;
            en8: in STD_LOGIC;
            en9: in STD_LOGIC;
            en10: in STD_LOGIC;
            en11: in STD_LOGIC;
            en12: in STD_LOGIC;
            result: out Signed (7 downto 0));
end shift_and_QU_DN;

architecture Behavioral of shift_and_QU_DN is

    component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                    p: out signed(15 downto 0)
       );
    end component;

    component ram_unit_block_2
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(2 downto 0);
                    write_addr: in std_logic_vector(2 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
    end component;

    component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                    sum: out signed(15 downto 0)
       );
    end component;

    component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                    clk: in std_logic;
                    en: in std_logic;
```

```
                rst: in std_logic
   );
end component;

component act_funct
   port ( result_in: in signed(15 downto 0);
          f_out: out signed(3 downto 0)
   );
end component;

component decoder_1_to_2
   port ( I: in signed(15 downto 0);
          S: in std_logic;
                O1: out signed(15 downto 0);
                O2: out signed(15 downto 0)
   );
end component;

component demux8
   port ( I: in std_logic_vector(7 downto 0);
          S: in std_logic;
                O1: out std_logic_vector(7 downto 0);
                O2: out std_logic_vector(7 downto 0)
   );
end component;

component mux2x1
    port ( I0: in signed(15 downto 0);
           I1: in signed(15 downto 0);
                    S0: in std_logic;
                    Y0: out signed(15 downto 0)
    );
end component;

component shitf_unit
    port ( x: in signed(7 downto 0);
           w: in signed(7 downto 0);
                    shift_out: out signed (15 downto 0)
    );
end component;

signal w_carrier: std_logic_vector(7 downto 0);              -- weights
signal w_carrier1: std_logic_vector(7 downto 0);             -- weights
signal w_carrier2: std_logic_vector(7 downto 0);             -- weights
signal w_carrier3: std_logic_vector(7 downto 0);             -- weights
signal w_carrier4: std_logic_vector(7 downto 0);             -- weights
signal w_carrier5: std_logic_vector(7 downto 0);             -- weights
signal w_carrier6: std_logic_vector(7 downto 0);             -- weights
signal w_carrier6_ex: std_logic_vector(15 downto 0);         -- weights
signal input_carrier1: std_logic_vector(7 downto 0); -- input to XOR unit
signal input_carrier2: std_logic_vector(7 downto 0); -- input to SH unit
signal prod_carrier: signed (15 downto 0);          -- output of xor unit
signal sum_carrier: signed(15 downto 0);            -- accumulated sum
signal partial_sum: signed(15 downto 0);            -- sum
signal f_out1: signed(15 downto 0);                 -- accumulated sum
signal f_out2: signed(15 downto 0);                 -- accumulated sum
signal prod_carrier5_red: signed(7 downto 0);
```

```
    signal f_out3: signed(3 downto 0);                              -- output
    signal prod_carrier1: signed(15 downto 0);        -- output of shift unit
    signal prod_carrier2: signed(15 downto 0);        -- weighted input
    signal prod_carrier3: signed(15 downto 0);        -- weighted input
    signal prod_carrier4: signed(15 downto 0);        -- weighted input
    signal prod_carrier5: signed(15 downto 0);        -- product carrier
    signal prod_carrier6: signed(15 downto 0);        -- product carrier
    signal sumA: signed(15 downto 0);                 -- accumulated sum A
    signal sumB: signed(15 downto 0);                 -- accumulated sum B
    signal partial_sum1: signed(15 downto 0);      -- hidden quantum output
    signal partial_sum2: signed(15 downto 0);               -- partial sum
    signal partial_sum3: signed(15 downto 0);               -- partial sum
    signal prod_A: signed(15 downto 0);                     -- result A
    signal carrier_resultA: signed(15 downto 0);         -- result A carrier
    signal prod_B: signed(15 downto 0);                  -- result B
    signal carrier_resultB: signed(15 downto 0);         -- result B carrier
    signal rst_carrier: std_logic;                       -- reset signal

begin

     Mem0: ram_unit_block_2 port map(data_in => input_w, read_addr =>
read_addr, write_addr => write_addr, en => we, clk => clk, data_out =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        DEM1: demux8 port map(I => w_carrier1, S => en2, O1 => w_carrier3, O2
=> w_carrier4);
        DEM2: demux8 port map(I => input_i, S => en2, O1 => input_carrier1,
O2 => input_carrier2);
        SHIFT_0: shitf_unit port map(x => signed(input_carrier1), w =>
signed(w_carrier3), shift_out => prod_carrier);
        Mul0: mul8x8 port map(x => signed(input_carrier2), y =>
signed(w_carrier4), p => prod_carrier1);
        MUX_0: mux2x1 port map(I0 => prod_carrier, I1 => prod_carrier1, S0 =>
en3, Y0 => prod_carrier2);
        Add0: add16 port map(a => prod_carrier2, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        Deco0: decoder_1_to_2 port map(I => sum_carrier, S => en4, O1 =>
prod_carrier3, O2 => prod_carrier4);
        Deco1: decoder_1_to_2 port map(I => prod_carrier3, S => en5, O1 =>
sumA, O2 => sumB);
        R1: reg16 port map(d => sumA, q => f_out1, clk => slow_clk, en =>
en6, rst => rst);
        R2: reg16 port map(d => sumB, q => f_out2, clk => slow_clk, en =>
en7, rst => rst);
        Add1: add16 port map(a => f_out1, b => f_out2, sum => partial_sum1);
        MUX_1: mux2x1 port map(I0 => partial_sum1, I1 => prod_carrier4, S0 =>
en8, Y0 => prod_carrier5);
        prod_carrier5_red <= prod_carrier5(7 downto 0);
        DEM3: demux8 port map(I => w_carrier2, S => en9, O1 => w_carrier5, O2
=> w_carrier6);
        Mul1: mul8x8 port map(x => prod_carrier5_red, y =>
signed(w_carrier5), p => prod_carrier6);
```

```
        Deco2: decoder_1_to_2 port map(I => prod_carrier6, S => en10, O1 =>
prod_A, O2 => prod_B);
        R3: reg16 port map(d => prod_A, q => carrier_resultA, clk =>
slow_clk, en => en11, rst => rst);
        R4: reg16 port map(d => prod_B, q => carrier_resultB, clk =>
slow_clk, en => en12, rst => rst);
        Add2: add16 port map(a => carrier_resultA, b => carrier_resultB, sum
=> partial_sum2);
        w_carrier6_ex <= "00000000" & w_carrier6;
        Add3: add16 port map(a => signed(w_carrier6_ex), b => partial_sum2,
sum => partial_sum3);
        act_f3: act_funct port map(result_in => partial_sum3, f_out =>
f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
```

C.9 Two XOR Units & 1 Quantum Neuron Model

```
----------------------------------------------------------------------------
-- Description:    DN composed of 2 XOR units & 1 quantum neuron
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity twoXOR_1QU is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          write_addr: in STD_LOGIC_VECTOR (2 downto 0);
          read_addr: in STD_LOGIC_VECTOR (2 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
          wt: in STD_LOGIC;
          slow_clk: in STD_LOGIC;
          en_acc: in STD_LOGIC;
          c_done: in std_logic;
          rst: in STD_LOGIC;
          en1: in STD_LOGIC;
          en2: in STD_LOGIC;
          en3: in STD_LOGIC;
          en4: in STD_LOGIC;
          en5: in STD_LOGIC;
          en6: in STD_LOGIC;
          en7: in STD_LOGIC;
          en8: in STD_LOGIC;
          en9: in STD_LOGIC;
          en10: in STD_LOGIC;
          en11: in STD_LOGIC;
          en12: in STD_LOGIC;
          result: out signed (7 downto 0));
end twoXOR_1QU;
```

```
architecture Behavioral of twoXOR_1QU is

  component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                   p: out signed(15 downto 0)
       );
  end component;

  component ram_unit_block_2
       port ( clk: std_logic;
              en: std_logic;
                   rd: std_logic;
                   wt: std_logic;
                   read_addr: in std_logic_vector(2 downto 0);
                   write_addr: in std_logic_vector(2 downto 0);
                   data_in: in std_logic_vector(7 downto 0);
                   data_out: out std_logic_vector(7 downto 0)
       );
  end component;

  component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                   sum: out signed(15 downto 0)
       );
  end component;

  component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                   clk: in std_logic;
                   en: in std_logic;
                   rst: in std_logic
       );
  end component;

  component act_funct
       port ( result_in: in signed(15 downto 0);
              f_out: out signed(3 downto 0)
       );
  end component;

  component decoder_1_to_2
       port ( I: in signed(15 downto 0);
              S: in std_logic;
                   O1: out signed(15 downto 0);
                   O2: out signed(15 downto 0)
       );
  end component;

  component demux8
       port ( I: in std_logic_vector(7 downto 0);
              S: in std_logic;
                   O1: out std_logic_vector(7 downto 0);
                   O2: out std_logic_vector(7 downto 0)
```

```
     );
  end component;

  component mux2x1
        port ( I0: in signed(15 downto 0);
               I1: in signed(15 downto 0);
                        S0: in std_logic;
                        Y0: out signed(15 downto 0)
        );
  end component;

  component xor_unit
        port ( x: in signed(7 downto 0);
               w: in signed(7 downto 0);
                        xor_out: out signed (15 downto 0)
        );
  end component;

  signal w_carrier: std_logic_vector(7 downto 0);                -- weights
  signal sum_carrier: signed(15 downto 0);                 -- accumulated sum
  signal w_carrier1: std_logic_vector(7 downto 0);         -- weights
  signal w_carrier2: std_logic_vector(7 downto 0);         -- weights
  signal prod_carrier: signed (15 downto 0);            -- output of xor units
  signal partial_sum: signed(15 downto 0);              -- sum
  signal f_out1: signed(15 downto 0);                   -- accumulated sum
  signal f_out3: signed(3 downto 0);                    -- output
  signal prod_carrier1: signed(15 downto 0);
  signal prod_carrier1_red: signed(7 downto 0);         -- reduced signal
  signal prod_carrier2: signed(15 downto 0);            -- weighted input
  signal prod_carrier3: signed(15 downto 0);            -- weighted input
  signal prod_carrier4: signed(15 downto 0);            -- weighted input
  signal prod_carrier5: signed(15 downto 0);            -- product carrier
  signal prod_carrier6: signed(15 downto 0);            -- product carrier
  signal A: signed(15 downto 0);                        -- accumulated sum A
  signal B: signed(15 downto 0);                        -- accumulated sum B
  signal partial_sum1: signed(15 downto 0);           -- hidden quantum output
  signal partial_sum2: signed(15 downto 0);                  -- partial sum
  signal prod_A: signed(15 downto 0);                        -- result A
  signal result_A: signed(15 downto 0);                      -- result A
  signal result_B: signed(15 downto 0);                      -- result B
  signal carrier_resultA: signed(15 downto 0);            -- result A carrier
  signal prod_B: signed(15 downto 0);                     -- result B
  signal carrier_resultB: signed(15 downto 0);            -- result B carrier
  signal rst_carrier: std_logic;                          -- reset signal

begin

     Mem0:  ram_unit_block_2  port  map(data_in  =>  input_w,  read_addr  =>
read_addr,  write_addr  =>  write_addr,  en  =>  we,  clk  =>  clk,  data_out  =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        XOR_0:    xor_unit    port    map(x    =>    signed(input_i),    w    =>
signed(w_carrier1), xor_out => prod_carrier);
        Add0: add16 port map(a => prod_carrier1, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
```

```
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        R1: reg16 port map(d => sum_carrier, q => f_out1, clk => slow_clk, en
=> en2, rst => rst);
        Deco0: decoder_1_to_2 port map(I => f_out1, S => en3, O1 => prod_A,
O2 => prod_B);
        R2: reg16 port map(d => prod_A, q => carrier_resultA, clk =>
slow_clk, en => en4, rst => rst);
        R3: reg16 port map(d => prod_B, q => carrier_resultB, clk =>
slow_clk, en => en5, rst => rst);
        MUX_0: mux2x1 port map(I0 => carrier_resultA, I1 => carrier_resultB,
S0 => en6, Y0 => prod_carrier1);
        prod_carrier1_red <= prod_carrier1(7 downto 0);
        Mul0:   mul8x8   port   map(x   =>   prod_carrier1_red,   y   =>
signed(w_carrier2), p => prod_carrier2);
        Deco1: decoder_1_to_2 port map(I => prod_carrier2, S => en7, O1 =>
prod_carrier3, O2 => prod_carrier4);
        R4: reg16 port map(d => prod_carrier3, q => result_A, clk =>
slow_clk, en => en8, rst => rst);
        R5: reg16 port map(d => prod_carrier4, q => result_B, clk =>
slow_clk, en => en9, rst => rst);
        Add1: add16 port map(a => result_A, b => result_B, sum =>
partial_sum1);
        Deco2: decoder_1_to_2 port map(I => partial_sum1, S => en10, O1 =>
prod_carrier5, O2 => prod_carrier6);
        R6: reg16 port map(d => prod_carrier5, q => A, clk => slow_clk, en =>
en11, rst => rst);
        R7: reg16 port map(d => prod_carrier6, q => B, clk => slow_clk, en =>
en12, rst => rst);
        Add3: add16 port map(a => A, b => B, sum => partial_sum2);
        act_f3: act_funct port map(result_in => partial_sum2, f_out =>
f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
```

C.10 Two Shift Units & 1 Quantum Neuron Model

```
----------------------------------------------------------------------------
-- Description:     DN composed of 2 Shift units and 1 Quantum neuron
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity twoshift_1QU is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          write_addr: in STD_LOGIC_VECTOR (2 downto 0);
          read_addr: in STD_LOGIC_VECTOR (2 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
```

```
            wt: in STD_LOGIC;
            slow_clk: in STD_LOGIC;
            en_acc: in STD_LOGIC;
            c_done: in STD_LOGIC;
            rst: in STD_LOGIC;
            en1: in STD_LOGIC;
            en2: in STD_LOGIC;
            en3: in STD_LOGIC;
            en4: in STD_LOGIC;
            en5: in STD_LOGIC;
            en6: in STD_LOGIC;
            en7: in STD_LOGIC;
            en8: in STD_LOGIC;
            en9: in STD_LOGIC;
            en10: in STD_LOGIC;
            en11: in STD_LOGIC;
            en12: in STD_LOGIC;
            result: out signed (7 downto 0));
end twoshift_1QU;

architecture Behavioral of twoshift_1QU is

  component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                    p: out signed(15 downto 0)
       );
  end component;

  component ram_unit_block_2
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(2 downto 0);
                    write_addr: in std_logic_vector(2 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
  end component;

  component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                    sum: out signed(15 downto 0)
       );
  end component;

  component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                    clk: in std_logic;
                    en: in std_logic;
                    rst: in std_logic
       );
  end component;
```

```
component act_funct
     port ( result_in: in signed(15 downto 0);
            f_out: out signed(3 downto 0)
     );
end component;

component decoder_1_to_2
     port ( I: in signed(15 downto 0);
            S: in std_logic;
                  O1: out signed(15 downto 0);
                  O2: out signed(15 downto 0)
     );
end component;

component demux8
     port ( I: in std_logic_vector(7 downto 0);
            S: in std_logic;
                  O1: out std_logic_vector(7 downto 0);
                  O2: out std_logic_vector(7 downto 0)
     );
end component;

component mux2x1
      port ( I0: in signed(15 downto 0);
             I1: in signed(15 downto 0);
                      S0: in std_logic;
                      Y0: out signed(15 downto 0)
      );
end component;

component shitf_unit
      port ( x: in signed(7 downto 0);
             w: in signed(7 downto 0);
                      shift_out: out signed (15 downto 0)
      );
end component;

signal w_carrier: std_logic_vector(7 downto 0);                 -- weights
signal sum_carrier: signed(15 downto 0);                -- accumulated sum
signal w_carrier1: std_logic_vector(7 downto 0);        -- weights
signal w_carrier2: std_logic_vector(7 downto 0);        -- weights
signal prod_carrier: signed (15 downto 0);          -- output of xor units
signal partial_sum: signed(15 downto 0);                -- sum
signal f_out1: signed(15 downto 0);                     -- accumulated sum
signal f_out3: signed(3 downto 0);                      -- output
signal prod_carrier1: signed(15 downto 0);                          --
signal prod_carrier1_red: signed(7 downto 0);           -- reduced signal
signal prod_carrier2: signed(15 downto 0);              -- weighted input
signal prod_carrier3: signed(15 downto 0);              -- weighted input
signal prod_carrier4: signed(15 downto 0);              -- weighted input
signal prod_carrier5: signed(15 downto 0);              -- product carrier
signal prod_carrier6: signed(15 downto 0);              -- product carrier
signal A: signed(15 downto 0);                       -- accumulated sum A
signal B: signed(15 downto 0);                       -- accumulated sum B
signal partial_sum1: signed(15 downto 0);          -- hidden quantum output
signal partial_sum2: signed(15 downto 0);          -- partial sum
signal prod_A: signed(15 downto 0);                -- result A
```

```
  signal result_A: signed(15 downto 0);                    -- result A
  signal result_B: signed(15 downto 0);                    -- result B
  signal carrier_resultA: signed(15 downto 0);           -- result A carrier
  signal prod_B: signed(15 downto 0);                    -- result B
  signal carrier_resultB: signed(15 downto 0);           -- result B carrier
  signal rst_carrier: std_logic;                           -- reset signal

begin

     Mem0: ram_unit_block_2 port map(data_in => input_w, read_addr => read_addr, write_addr => write_addr, en => we, clk => clk, data_out => w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2 => w_carrier2);
        shitf_0: shitf_unit port map(x => signed(input_i), w => signed(w_carrier1), shift_out => prod_carrier);
        Add0: add16 port map(a => prod_carrier1, b => sum_carrier, sum => partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk => slow_clk, en => en_acc, rst => rst_carrier);
        R1: reg16 port map(d => sum_carrier, q => f_out1, clk => slow_clk, en => en2, rst => rst);
        Deco0: decoder_1_to_2 port map(I => f_out1, S => en3, O1 => prod_A, O2 => prod_B);
        R2: reg16 port map(d => prod_A, q => carrier_resultA, clk => slow_clk, en => en4, rst => rst);
        R3: reg16 port map(d => prod_B, q => carrier_resultB, clk => slow_clk, en => en5, rst => rst);
        MUX_0: mux2x1 port map(I0 => carrier_resultA, I1 => carrier_resultB, S0 => en6, Y0 => prod_carrier1);
        prod_carrier1_red <= prod_carrier1(7 downto 0);
        Mul0: mul8x8 port map(x => prod_carrier1_red, y => signed(w_carrier2), p => prod_carrier2);
        Deco1: decoder_1_to_2 port map(I => prod_carrier2, S => en7, O1 => prod_carrier3, O2 => prod_carrier4);
        R4: reg16 port map(d => prod_carrier3, q => result_A, clk => slow_clk, en => en8, rst => rst);
        R5: reg16 port map(d => prod_carrier4, q => result_B, clk => slow_clk, en => en9, rst => rst);
        Add1: add16 port map(a => result_A, b => result_B, sum => partial_sum1);
        Deco2: decoder_1_to_2 port map(I => partial_sum1, S => en10, O1 => prod_carrier5, O2 => prod_carrier6);
        R6: reg16 port map(d => prod_carrier5, q => A, clk => slow_clk, en => en11, rst => rst);
        R7: reg16 port map(d => prod_carrier6, q => B, clk => slow_clk, en => en12, rst => rst);
        Add3: add16 port map(a => A, b => B, sum => partial_sum2);
        act_f3: act_funct port map(result_in => partial_sum2, f_out => f_out3);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
```

C.11 Two Quantum Neurons & 1 XOR Unit Model

```
----------------------------------------------------------------------------
-- Description:    DN composed of two quantum neurons and 1 XOR unit
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity twoQU_1XOR is
    Port ( input_i : in  STD_LOGIC_VECTOR (7 downto 0);
           input_w : in  STD_LOGIC_VECTOR (7 downto 0);
           read_addr : in  STD_LOGIC_VECTOR (2 downto 0);
           write_addr : in  STD_LOGIC_VECTOR (2 downto 0);
           we : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           rd : in  STD_LOGIC;
           wt : in  STD_LOGIC;
           slow_clk : in  STD_LOGIC;
           en_acc : in  STD_LOGIC;
           c_done: in std_logic;
           rst : in  STD_LOGIC;
           en1 : in  STD_LOGIC;
           en2 : in  STD_LOGIC;
           en3 : in  STD_LOGIC;
           en4 : in  STD_LOGIC;
           en5 : in  STD_LOGIC;
           en6 : in  STD_LOGIC;
           en7 : in  STD_LOGIC;
           result : out  signed (7 downto 0));
end twoQU_1XOR;

architecture Behavioral of twoQU_1XOR is

    component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                    p: out signed(15 downto 0)
       );
    end component;

    component ram_unit_block_2
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(2 downto 0);
                    write_addr: in std_logic_vector(2 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
    end component;

    component add16
       port ( a: in signed(15 downto 0);
```

```
          b: in signed(15 downto 0);
          sum: out signed(15 downto 0)
   );
 end component;

 component reg16
   port (d: in signed(15 downto 0);
          q: out signed(15 downto 0);
                clk: in std_logic;
                en: in std_logic;
                rst: in std_logic
   );
 end component;

component act_funct
   port ( result_in: in signed(15 downto 0);
          f_out: out signed(3 downto 0)
   );
end component;

component decoder_1_to_2
   port ( I: in signed(15 downto 0);
          S: in std_logic;
                O1: out signed(15 downto 0);
                O2: out signed(15 downto 0)
   );
end component;

component demux8
   port ( I: in std_logic_vector(7 downto 0);
          S: in std_logic;
                O1: out std_logic_vector(7 downto 0);
                O2: out std_logic_vector(7 downto 0)
   );
end component;

component xor_unit
    port ( x: in signed(7 downto 0);
           w: in signed(7 downto 0);
                    xor_out: out signed (15 downto 0)
    );
end component;

signal w_carrier: std_logic_vector(7 downto 0);              -- weights
signal w_carrier1: std_logic_vector(7 downto 0);             -- weights
signal w_carrier2: std_logic_vector(7 downto 0);             -- weights
signal prod_carrier: signed (15 downto 0);                 -- product output
signal sum_carrier: signed(15 downto 0);                  -- accumulated sum
signal partial_sum: signed(15 downto 0);                  -- sum
signal rst_carrier: std_logic;                            -- reset signal
signal prod_A: signed(15 downto 0);                       -- result A
signal prod_B: signed(15 downto 0);                       -- result B
signal prodA_carrier: signed(15 downto 0);               -- product A carrier
signal prodB_carrier: signed(15 downto 0);               -- product B carrier
signal partial_sum1: signed(15 downto 0);
signal partial_sum1_red: signed(7 downto 0);
    signal prod_carrier1: signed(15 downto 0);
```

```
    signal result_A: signed(15 downto 0);                    -- result A
    signal result_B: signed(15 downto 0);                    -- result B
    signal resultA_carrier: signed(15 downto 0);
    signal resultB_carrier: signed(15 downto 0);
    signal f_out: signed(15 downto 0);
    signal f_out3: signed(3 downto 0);                       -- output

begin

     Mem0: ram_unit_block_2 port map(data_in => input_w, read_addr => read_addr, write_addr => write_addr, en => we, clk => clk, data_out => w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2 => w_carrier2);
        Mul0: mul8x8 port map(x => signed(input_i), y => signed(w_carrier1), p => prod_carrier);
        Add0: add16 port map(a => prod_carrier, b => sum_carrier, sum => partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk => slow_clk, en => en_acc, rst => rst_carrier);
        Deco0: decoder_1_to_2 port map(I => sum_carrier, S => en2, O1 => prod_A, O2 => prod_B);
        R1: reg16 port map(d => prod_A, q => prodA_carrier, clk => slow_clk, en => en3, rst => rst);
        R2: reg16 port map(d => prod_B, q => prodB_carrier, clk => slow_clk, en => en4, rst => rst);
        Add1: add16 port map(a => prodA_carrier, b => prodB_carrier, sum => partial_sum1);
        partial_sum1_red <= partial_sum1(7 downto 0);
        XOR_0: xor_unit port map(x => partial_sum1_red, w => signed(w_carrier2), xor_out => prod_carrier1);
        Deco1: decoder_1_to_2 port map(I => prod_carrier1, S => en5, O1 => result_A, O2 => result_B);
        R3: reg16 port map(d => result_A, q => resultA_carrier, clk => slow_clk, en => en6, rst => rst);
        R4: reg16 port map(d => result_B, q => resultB_carrier, clk => slow_clk, en => en7, rst => rst);
        Add2: add16 port map(a => resultA_carrier, b => resultB_carrier, sum => f_out);
        f_out3 <= f_out(3 downto 0);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
```

C.12 Two Quantum Neurons & 1 Shift Unit Model

```
----------------------------------------------------------------------------
-- Description:    1 DN composed of 2 quantum neurons and 1 shift unit
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;
```

```
entity twoQU_1shift is
    Port ( input_i : in  STD_LOGIC_VECTOR (7 downto 0);
           input_w : in  STD_LOGIC_VECTOR (7 downto 0);
           read_addr : in  STD_LOGIC_VECTOR (2 downto 0);
           write_addr : in  STD_LOGIC_VECTOR (2 downto 0);
           we : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           rd : in  STD_LOGIC;
           wt : in  STD_LOGIC;
           slow_clk : in  STD_LOGIC;
           en_acc : in  STD_LOGIC;
           c_done: in std_logic;
           rst : in  STD_LOGIC;
           en1 : in  STD_LOGIC;
           en2 : in  STD_LOGIC;
           en3 : in  STD_LOGIC;
           en4 : in  STD_LOGIC;
           en5 : in  STD_LOGIC;
           en6 : in  STD_LOGIC;
           en7 : in  STD_LOGIC;
           result : out signed (7 downto 0));
end twoQU_1shift;

architecture Behavioral of twoQU_1shift is

     component mul8x8
       port ( x: in signed(7 downto 0);
              y: in signed(7 downto 0);
                    p: out signed(15 downto 0)
       );
       end component;

       component ram_unit_block_2
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(2 downto 0);
                    write_addr: in std_logic_vector(2 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
       end component;

       component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                    sum: out signed(15 downto 0)
       );
       end component;

       component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                    clk: in std_logic;
                    en: in std_logic;
                    rst: in std_logic
```

```
   );
end component;

component act_funct
   port ( result_in: in signed(15 downto 0);
          f_out: out signed(3 downto 0)
   );
end component;

component decoder_1_to_2
   port ( I: in signed(15 downto 0);
          S: in std_logic;
                O1: out signed(15 downto 0);
                O2: out signed(15 downto 0)
   );
end component;

component demux8
   port ( I: in std_logic_vector(7 downto 0);
          S: in std_logic;
                O1: out std_logic_vector(7 downto 0);
                O2: out std_logic_vector(7 downto 0)
   );
end component;

component shitf_unit
    port ( x: in signed(7 downto 0);
           w: in signed(7 downto 0);
                    shift_out: out signed (15 downto 0)
    );
end component;

signal w_carrier: std_logic_vector(7 downto 0);          -- weights
signal w_carrier1: std_logic_vector(7 downto 0);         -- weights
signal w_carrier2: std_logic_vector(7 downto 0);         -- weights
signal prod_carrier: signed (15 downto 0);          -- product output
signal sum_carrier: signed(15 downto 0);            -- accumulated sum
signal partial_sum: signed(15 downto 0);                        -- sum
signal rst_carrier: std_logic;                      -- reset signal
signal prod_A: signed(15 downto 0);                 -- result A
signal prod_B: signed(15 downto 0);                 -- result B
signal prodA_carrier: signed(15 downto 0);          -- product A carrier
signal prodB_carrier: signed(15 downto 0);          -- product B carrier
signal partial_sum1: signed(15 downto 0);
signal partial_sum1_red: signed(7 downto 0);
signal prod_carrier1: signed(15 downto 0);
signal result_A: signed(15 downto 0);               -- result A
signal result_B: signed(15 downto 0);               -- result B
signal resultA_carrier: signed(15 downto 0);
signal resultB_carrier: signed(15 downto 0);
signal f_out: signed(15 downto 0);
signal f_out3: signed(3 downto 0);                          -- output

begin
```

```
     Mem0:  ram_unit_block_2  port  map(data_in  =>  input_w,  read_addr  =>
read_addr,  write_addr  =>  write_addr,  en  =>  we,  clk  =>  clk,  data_out  =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        Mul0: mul8x8 port map(x => signed(input_i), y => signed(w_carrier1),
p => prod_carrier);
        Add0: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        Deco0: decoder_1_to_2 port map(I => sum_carrier, S => en2, O1 =>
prod_A, O2 => prod_B);
        R1: reg16 port map(d => prod_A, q => prodA_carrier, clk => slow_clk,
en => en3, rst => rst);
        R2: reg16 port map(d => prod_B, q => prodB_carrier, clk => slow_clk,
en => en4, rst => rst);
        Add1: add16 port map(a => prodA_carrier, b => prodB_carrier, sum =>
partial_sum1);
        partial_sum1_red <= partial_sum1(7 downto 0);
        SHIFT_0:   shitf_unit   port   map(x   =>   partial_sum1_red,   w   =>
signed(w_carrier2), shift_out => prod_carrier1);
        Deco1: decoder_1_to_2 port map(I => prod_carrier1, S => en5, O1 =>
result_A, O2 => result_B);
        R3: reg16 port map(d => result_A, q => resultA_carrier, clk =>
slow_clk, en => en6, rst => rst);
        R4: reg16 port map(d => result_B, q => resultB_carrier, clk =>
slow_clk, en => en7, rst => rst);
        Add2: add16 port map(a => resultA_carrier, b => resultB_carrier, sum
=> f_out);
        f_out3 <= f_out(3 downto 0);
        -- output
        result <= "0000" & f_out3;
end Behavioral;
```

C.13 3-XOR Units

```
----------------------------------------------------------------------------
-- Description:    DN composed of 3 XOR units.
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity threeXOR_DN is
    Port (input_i: in STD_LOGIC_VECTOR (7 downto 0);
          input_w: in STD_LOGIC_VECTOR (7 downto 0);
          read_addr: in STD_LOGIC_VECTOR (1 downto 0);
          write_addr: in STD_LOGIC_VECTOR (1 downto 0);
          we: in STD_LOGIC;
          clk: in STD_LOGIC;
          rd: in STD_LOGIC;
```

```
           wt: in STD_LOGIC;
           slow_clk: in STD_LOGIC;
           en_acc: in STD_LOGIC;
           c_done: in STD_LOGIC;
           rst: in STD_LOGIC;
           en1: in STD_LOGIC;
           en2: in STD_LOGIC;
           en3: in STD_LOGIC;
           en4: in STD_LOGIC;
           en5: in STD_LOGIC;
           result: out signed (7 downto 0));
end threeXOR_DN;

architecture Behavioral of threeXOR_DN is

       component ram_unit_block
       port ( clk: std_logic;
              en: std_logic;
                     rd: std_logic;
                     wt: std_logic;
                     read_addr: in std_logic_vector(1 downto 0);
                     write_addr: in std_logic_vector(1 downto 0);
                     data_in: in std_logic_vector(7 downto 0);
                     data_out: out std_logic_vector(7 downto 0)
       );
       end component;

       component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                     sum: out signed(15 downto 0)
       );
       end component;

       component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                     clk: in std_logic;
                     en: in std_logic;
                     rst: in std_logic
       );
       end component;

       component decoder_1_to_2
       port ( I: in signed(15 downto 0);
              S: in std_logic;
                     O1: out signed(15 downto 0);
                     O2: out signed(15 downto 0)
       );
       end component;

       component demux8
       port ( I: in std_logic_vector(7 downto 0);
              S: in std_logic;
                     O1: out std_logic_vector(7 downto 0);
                     O2: out std_logic_vector(7 downto 0)
       );
```

```
        end component;

        component xor_unit
         port ( x: in signed(7 downto 0);
                w: in signed(7 downto 0);
                        xor_out: out signed (15 downto 0)
         );
        end component;

        signal w_carrier: std_logic_vector(7 downto 0);             -- weights
        signal w_carrier1: std_logic_vector(7 downto 0);            -- weights
        signal w_carrier2: std_logic_vector(7 downto 0);            -- weights
        signal prod_carrier: signed (15 downto 0);       -- output of xor unit
        signal sum_carrier: signed(15 downto 0);         -- accumulated sum
        signal partial_sum: signed(15 downto 0);         -- result from adder
        signal f_out1: signed(15 downto 0);          -- accumulated sum carrier
        signal prod_carrier1: signed(15 downto 0);-- output of second XOR unit
        signal prod_A: signed(15 downto 0);                    -- result A
        signal prod_B: signed(15 downto 0);                    -- result B
        signal rst_carrier: std_logic;                         -- reset signal
        signal carrier_resultA: signed(15 downto 0);       -- result A carrier
        signal carrier_resultB: signed(15 downto 0);       -- result B carrier
        signal partial_sum1: signed(15 downto 0);
        signal f_out1_red: signed(7 downto 0);             -- output signal
        signal f_out3_red: signed(7 downto 0);             -- output signal

begin

     Mem0:  ram_unit_block  port  map(data_in  =>  input_w,  read_addr  =>
read_addr,  write_addr  =>  write_addr,  en  =>  we,  clk  =>  clk,  data_out  =>
w_carrier, rd => rd, wt => wt);
        DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
        XOR_0:   xor_unit   port   map(x   =>   signed(input_i),   w   =>
signed(w_carrier1), xor_out => prod_carrier);
        Add0: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);
        rst_carrier <= c_done or rst;
        R0:  reg16  port  map(d  =>  partial_sum,  q  =>  sum_carrier,  clk  =>
slow_clk, en => en_acc, rst => rst_carrier);
        R1: reg16 port map(d => sum_carrier, q => f_out1, clk => slow_clk, en
=> en2, rst => rst);
        f_out1_red <= f_out1(7 downto 0);
        XOR_1: xor_unit port map(x => f_out1_red, w => signed(w_carrier2),
xor_out => prod_carrier1);
        Deco0: decoder_1_to_2 port map(I => prod_carrier1, S => en3, O1 =>
prod_A, O2 => prod_B);
        R2:  reg16  port  map(d  =>  prod_A,  q  =>  carrier_resultA,  clk  =>
slow_clk, en => en4, rst => rst);
        R3:  reg16  port  map(d  =>  prod_B,  q  =>  carrier_resultB,  clk  =>
slow_clk, en => en5, rst => rst);
        Add1: add16 port map(a => carrier_resultA, b => carrier_resultB, sum
=> partial_sum1);
        f_out3_red <= partial_sum1(7 downto 0);
        -- output
        result <= f_out3_red;
end Behavioral;
```

C.14 3-Shift Units

```
----------------------------------------------------------------------------
-- Description:    DN composed of 3 shift units.
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity threeshift_DN is
    Port ( input_i : in  STD_LOGIC_VECTOR (7 downto 0);
           input_w : in  STD_LOGIC_VECTOR (7 downto 0);
           read_addr : in  STD_LOGIC_VECTOR (1 downto 0);
           write_addr : in  STD_LOGIC_VECTOR (1 downto 0);
           we : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           rd : in  STD_LOGIC;
           wt : in  STD_LOGIC;
           slow_clk : in  STD_LOGIC;
           en_acc : in  STD_LOGIC;
           c_done : in  STD_LOGIC;
           rst:in std_logic;
           en1 : in  STD_LOGIC;
           en2 : in  STD_LOGIC;
           en3 : in  STD_LOGIC;
           en4 : in  STD_LOGIC;
           en5 : in  STD_LOGIC;
           result : out  signed (7 downto 0));
end threeshift_DN;

architecture Behavioral of threeshift_DN is

    component ram_unit_block
       port ( clk: std_logic;
              en: std_logic;
                    rd: std_logic;
                    wt: std_logic;
                    read_addr: in std_logic_vector(1 downto 0);
                    write_addr: in std_logic_vector(1 downto 0);
                    data_in: in std_logic_vector(7 downto 0);
                    data_out: out std_logic_vector(7 downto 0)
       );
    end component;

    component add16
       port ( a: in signed(15 downto 0);
              b: in signed(15 downto 0);
                    sum: out signed(15 downto 0)
       );
    end component;

    component reg16
       port (d: in signed(15 downto 0);
              q: out signed(15 downto 0);
                    clk: in std_logic;
```

```
                en: in std_logic;
                rst: in std_logic
    );
end component;

component decoder_1_to_2
   port ( I: in signed(15 downto 0);
          S: in std_logic;
                O1: out signed(15 downto 0);
                O2: out signed(15 downto 0)
    );
end component;

component demux8
   port ( I: in std_logic_vector(7 downto 0);
          S: in std_logic;
                O1: out std_logic_vector(7 downto 0);
                O2: out std_logic_vector(7 downto 0)
    );
end component;

component shitf_unit
    port ( x: in signed(7 downto 0);
           w: in signed(7 downto 0);
                    shift_out: out signed (15 downto 0)
    );
end component;

signal w_carrier: std_logic_vector(7 downto 0);            -- weights
signal w_carrier1: std_logic_vector(7 downto 0);           -- weights
signal w_carrier2: std_logic_vector(7 downto 0);           -- weights
signal prod_carrier: signed (15 downto 0);          -- output of xor unit
signal sum_carrier: signed(15 downto 0);            -- accumulated sum
signal partial_sum: signed(15 downto 0);            -- result from adder
signal f_out1: signed(15 downto 0);             -- accumulated sum carrier
signal prod_carrier1: signed(15 downto 0);   -- output of second XOR unit
signal prod_A: signed(15 downto 0);                          -- result A
signal prod_B: signed(15 downto 0);                          -- result B
signal rst_carrier: std_logic;                             -- reset signal
signal carrier_resultA: signed(15 downto 0);          -- result A carrier
signal carrier_resultB: signed(15 downto 0);          -- result B carrier
signal partial_sum1: signed(15 downto 0);
signal f_out1_red: signed(7 downto 0);                -- output signal
signal f_out3_red: signed(7 downto 0);                -- output signal

begin

     Mem0:  ram_unit_block  port  map(data_in  =>  input_w,  read_addr  =>
read_addr,  write_addr  =>  write_addr,  en  =>  we,  clk  =>  clk,  data_out  =>
w_carrier, rd => rd, wt => wt);
       DEM0: demux8 port map(I => w_carrier, S => en1, O1 => w_carrier1, O2
=> w_carrier2);
       XOR_0:   shitf_unit   port   map(x   =>   signed(input_i),   w   =>
signed(w_carrier1), shift_out => prod_carrier);
       Add0: add16 port map(a => prod_carrier, b => sum_carrier, sum =>
partial_sum);
       rst_carrier <= c_done or rst;
```

```
        R0: reg16 port map(d => partial_sum, q => sum_carrier, clk =>
slow_clk, en => en_acc, rst => rst_carrier);
        R1: reg16 port map(d => sum_carrier, q => f_out1, clk => slow_clk, en
=> en2, rst => rst);
        f_out1_red <= f_out1(7 downto 0);
        XOR_1: shitf_unit port map(x => f_out1_red, w => signed(w_carrier2),
shift_out => prod_carrier1);
        Deco0: decoder_1_to_2 port map(I => prod_carrier1, S => en3, O1 =>
prod_A, O2 => prod_B);
        R2: reg16 port map(d => prod_A, q => carrier_resultA, clk =>
slow_clk, en => en4, rst => rst);
        R3: reg16 port map(d => prod_B, q => carrier_resultB, clk =>
slow_clk, en => en5, rst => rst);
        Add1: add16 port map(a => carrier_resultA, b => carrier_resultB, sum
=> partial_sum1);
        f_out3_red <= partial_sum1(7 downto 0);
        -- output
        result <= f_out3_red;
end Behavioral;
```

CURRICULUM VITAE

Jovan Saenz was born in Chihuahua, Chihuahua Mexico. He is the oldest son of Jorge Saenz and Guadalupe Lopez. His wife is Angeles Villar. He received his Bachelor's degree in Electrical Engineering in May 2001. In July, 2003, he received his Master's degree in Electrical Engineering from the same institution. His job experience includes internships at GCC Cemento and Applied Materials. He has held full time positions in the automotive industry at Delphi Automotive Systems as Web Applications Developer & Programmer, Automation & Controls Engineer, and Hardware & Software Engineer.

E-Mail: jsaenz@miners.utep.edu