This is to certify that the

thesis entitled

FUNCTIONAL CORROBORATION OF DIGITAL MULTILAYER NEURAL NETWORKS

presented by

QAISER HAMEED MALIK

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng

Major professor

Date June 21, 1995

FUNCTIONAL CORROBORATION OF A DIGITAL MULTILAYER NEURAL NETWORK

By

Qaiser Hameed Malik

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical Engineering

1995

ABSTRACT

FUNCTIONAL CORROBORATION OF A DIGITAL MULTILAYER NEURAL NETWORK

By

Qaiser Hameed Malik

A model of the digital multilayer neural network (DMNN), has been implemented in software using the C programming language. The DMNN employs a stochastic nonlinear function in the backpropagation learning rule. A suite of test cases is applied to the simulated DMNN and its performance is analyzed with reference to an ordinary Artificial Neural Network (ANN) employing a sigmoidal function. The backpropagation learning algorithm, used in the DMNN, has been revised in order to accelerate the convergence process. A modified DMNN has been simulated based upon the refinements suggested in the backpropagation algorithm. The modified DMNN has been found to converge for a wider range of learning rates (η) and momentum factors (α).

Three networks, DMNN, modified DMNN and ANN, are simulated on an 80486 based Personal Computer (PC). The learning pattern of each network is determined for the test cases and their performance is analyzed.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Professor Michael A. Shanblatt for his guidance, patience and encouragement throughout the course of this research. I would also like to thank Dr. Niaz Mahmood and Dr. Muhammad Akbar, my co-advisors at the National University of Sciences and Technology (Pakistan), for their continued support and supervision. Thanks are also due to the members of my Master's defence committee, Dr. Mohammad Aslam and Dr. Diane Rover for their comments and valuable suggestions.

My sincere and heartfelt thanks are due to my friend Abdul Yamin for providing an understanding of the programming techniques in C language and assisting me in the laborious task of network simulations.

I would like to express my gratitude and appreciation to my mother, my wife and my children, Farina and Arsalan, for their continuous love, long patience and understanding during the course of this research.

I am indebted to the National University of Sciences and Technology (Pakistan), for providing me an opportunity to pursue my MS degree program at the Michigan State University, USA.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Artificial Neural Networks (ANN) are systems made of a large number of simple neurons or Processing Elements (PE), whose function is determined mainly by their interconnection topology. These systems are capable of solving computationally intensive and ill-defined problems such as pattern classification, adaptive control, optimization and many more data processing tasks. Currently there are many models available that can be implemented via software and hardware. Software implementation usually employs an algorithm, which is based on the architecture and an understanding of neural networks, mapped onto the conventional digital computers. This thesis presents a software model of a pulse mode Digital Multilayer Neural Network (DMNN) developed by Kim [1]. The capabilities of the network are validated and compared with an ordinary ANN by using data encoding and pattern classification problems.

This chapter begins with a brief overview of the DMNN model. The problem to be solved is then defined, followed by the research tasks. Finally the organization of this thesis is outlined.

1.1 Overview

Conventional digital computers are very useful in solving well-defined problems since they can be represented by sequences of instructions. However, conventional computers can not compete with the performance of human beings in solving ill-defined random problems such as pattern recognition, classification, speech understanding, vision and so on. The typical characteristics of human nervous system and the brain are; adaptive learning, generalization, error correction, robustness, and creativity. Its computational power is due to its massive interconnection and asynchronous parallel communication among neurons (basic anatomical units of the nervous system).

Biologically inspired computing machines, often referred to as artificial neural networks (ANN), work in a way similar to the neurons in the human brain. In the past few years, neural networks have received a great deal of attention and are being touted as one of the important computational tools developed so far. Much of the excitement is due to the apparent ability of the neural networks to imitate the brain's ability to make decisions and draw conclusions when presented with complex, noisy, irrelevant and/or partial information. It is believed that the massive parallelism and computational power of the human brain is due to the complex interconnections among a large number of neurons and not due to the complexity of an individual neuron. Artificial neural networks are inspired both by neurobiology and various mathematical theories of learning and information processing. The main objective of the ANN research is to design new

architectures and algorithms that can solve problems which are difficult to solve by conventional digital computers.

With the enormous developments in the fields of VLSI technology and with improved understanding of the human nervous system, it has become possible to implement models of neural nets by mimicking some aspects of the nervous system of the mammals. The implementation of artificial neural networks is based on a large number of simple computational elements. Currently there are many models of ANNs [2-4] that can be implemented via hardware or software. Software implementation usually employs an algorithm which is based on the architecture and an understanding of the neural networks, mapped onto the conventional digital computers. Software implementation is very flexible since one can easily modify codes or algorithms in order to implement another ANN model or learning mechanism. Hardware implementation is more suitable to implementing large ANNs and its computational speed is much faster than that of the software implementation. With the current advancements in VLSI technology, dedicated hardware implementation of ANNs is progressing and many analog and digital ANN models have been built [5-8].

Recently, a new digital approach has been introduced in which a synaptic multiplication and neuron activation function is implemented with the simple logic gates using stochastic computing techniques [9,10]. Based on this technique, a statistical model of a pulse mode Digital Multilayer Neural Network (DMNN) has been developed [1]. The applicability of the developed network has been demonstrated using benchmark comparisons and character recognition problems.

This thesis presents the functional validation of the DMNN presented by Kim [1]. A software simulation of the DMNN is presented and its results are compared with an ordinary ANN. The backpropagation algorithm, used in the DMNN, is modified to accelerate the convergence. Based on these modifications, an improved DMNN is presented and its performance is evaluated by solving a suite of test cases.

1.2 Problem Statement

Many researchers have worked on the implementation of artificial neural networks. Consequently a number of ANN models have been developed both in software and hardware. Software implementation offers more flexibility in carrying out the performance evaluation of various ANN models. However, the software implementation suffers from the major disadvantages of excessive computational time and increased memory requirements with increase in the number of neurons (processing elements). Hardware ANN implementation models have been built using analog and digital components [10-13]. Analog implementation due to its inherent drawbacks, such as, inaccuracy of analog computations and low design flexibility of the analog electronic devices, limit their chip density thus restricting their applications.

The advanced design techniques of current VLSI technology have facilitated the digital implementation of the ANNs. To reduce the area requirements of a conventional digital approach, stochastic computing techniques have been developed that have resulted in the possibility of a low cost and high speed digital ANN implementation. In these

architectures, algebraic operations are replaced by random processes and the network performs pseudo-analog computations with operands ranging from 0.0 to 1.0. An operand x in the pulse mode representation is the probability of pulse occurrence in the corresponding binary pseudo-random pulse sequence $x_{(n)}$, generated at each clock [9,10].

Based on this technique, a pulse mode digital multilayer neural network (DMNN) architecture and its corresponding statistical model has been developed by Kim [1]. Kim has employed the VHSIC (Very High Speed Integrated Circuits) Hardware Description Language (VHDL) for simulating the DMNN [1]. However, there is a need to verify the applicability of the developed network using software simulation techniques. The DMNN needs to be compared with some standard ANN model to observe the performance of the stochastic function used in the backpropagation algorithm.

1.3 Research Tasks

The research tasks identified for this work are:

1. Simulate the statistical model of the digital multilayer neural network (DMNN), developed by Kim [1], in software using the C programming language.
2. Apply a suite of test cases to the simulated DMNN, and verify its results.
3. Verify validity of the DMNN by solving the test cases with a conventional ANN using a sigmoidal function and comparing their results.

To simulate the statistical model of the DMNN, the first step is to understand the model developed by Kim [1]. The research is, therefore, initiated with the study of artificial neural networks. An understanding of the various models of ANNs, based on different learning algorithms, is developed with a special emphasis on the backpropagation algorithm. A detailed study of the statistical model of the DMNN developed by Kim, is undertaken. The DMNN is simulated in C, using a 80486 based Personal Computer (PC).

A suite of test problems is applied to the DMNN and the results achieved by Kim are validated. The DMNN is first applied to the famous XOR problem. Next, an 8x3 data encoder is simulated and the results compiled in graphical form. Efforts are then directed to simulate the character classification problem. Two experiments are attempted: one with a 5-digit classifier and the other with a 10-digit classifier. Each experiment is first performed with an ideal set of digits and then with an addition of noisy data.

For the last task, an artificial neural network (ANN) using an ordinary sigmoidal function is simulated using C language. The network is presented with the same set of test problems and results are compared with the performance of the DMNN.

A number of research constraints have been encountered that have restricted the verification of the character classification problem. The proficiency of the simulated DMNN is enhanced by incorporating a number of variations in the program structure and its logic. The backpropagation algorithm is modified in an attempt to simulate the character classifier given the limitations of the PC.

1.4 Organization of the Thesis

This thesis has been organized in six chapters. Chapter 2 discusses the background of artificial neural networks. It begins with a brief history of the different development phases and the important contributions made by various researchers and scientists in this field. Working models of biological and artificial neurons are presented and basic building blocks of an artificial neural network are introduced. Various ANN models are presented and their learning rules are defined. The chapter concludes with a discussion of few implementation techniques.

Chapter 3 is a summary/review of Kim's work [1]. The digital multilayer neural network developed by Kim is presented. The stochastic computing techniques are introduced and the nonlinear neuron activation function used in the DMNN is presented. Basic computing elements of the DMNN are introduced and a modular DMNN architecture is discussed. The DMNN is applied to a suite of test cases and its performance is presented.

Chapter 4 discusses the implementation of the backpropagation algorithm in the DMNN. A complete mathematical derivation of the feedforward and feedback processes is presented. A two layer feedforward DMNN is simulated in C using the backpropagation learning rule. The backpropagation learning rule is analyzed for its weaknesses and improvements are suggested to accelerate the convergence process.

The DMNN simulation results are presented in Chapter 5. The results of the application problems are compared to those attained through a standard ANN using a sigmoidal function. The improvements suggested in Chapter 4 are then incorporated to enhance the performance of the DMNN. The results are tabulated in graphical form for ease of comparison.

Finally, Chapter 6 lists major contributions of the research and identifies future directions of this work.

CHAPTER 2

BACKGROUND

2.1 Introduction

The human brain is the most complex computing device known to man. The brain's powerful thinking, remembering, and problem-solving capabilities have inspired many scientists to attempt modeling of its operation. One group of researchers has sought to create a computer model that matches the functionality of the brain in a very simplified manner, the result has been the study of Artificial Neural Networks (ANN).

This chapter discusses the brief history of artificial neural networks. A review of biological and artificial neurons is provided. Next, various learning rules and typical network models based on these learning rules, with a special emphasis on the feedforward multilayer perceptron using backpropagation learning, are discussed. Thereafter, different implementation techniques currently employed are introduced.

2.2 Historical Background

This section briefly discusses the history of artificial neural networks. From now onwards the artificial neural networks will be referred to as neural networks, neural nets or ANNs.

Attempts to understand the human brain go back a long way, even centuries. It has been almost exactly a century, 1890, since William James's text, Psychology (Briefer Course) provided insights into brain activity and foreshadowed current theories [14]. Authors of many texts and papers quote the following from William James's text as an example:

> *" Let us then assume as the basis of all our subsequent reasoning this law: When two elementary brain processes have been active together or in immediate succession, one of them, on re-occurring, tends to propagate its excitement into the other. "*

In 1936, Alan Turing was the first to use the brain as computing example. In 1943, McCulloch and Pitts published one of the most famous neural network papers about how neurons might work [15]. McCulloch and Pitts derived models of neural networks based on what was known about the biological structures in the early 1940s. In coming to their conclusions, they stated five physical assumptions:

1. The activity of a neuron is an 'all-or-none' process.

2. A certain fixed number of synapses must be excited within a period of time during which the neuron is able to detect the values present on its inputs, the synapses, in order to excite a neuron at any time. This number is independent of previous activity on the neuron.

3. The only significant delay within the nervous system is synaptic delay, *i.e.*, the time delay between sensing inputs and acting on them by transmitting an outgoing pulse.

4. The activity of an inhibitory synapse absolutely prevents excitation of the neuron at that time.

5. The structure of the net does not change with time.

The neuron described by these five assumptions is known as McCulloch-Pitts neuron. The next personality in the Age of Conception is Donald O. Hebb, whose 1949 book was the first one to define the method of updating synaptic weights that we now refer to as Hebbian [16]. Donald O. Hebb is among the first to use the term 'connectionism'. The 1950s was the age of computer simulation. It became possible to test theories about nervous system functions. The IBM research laboratories conducted an unsuccessful software simulation of a neural network model based on Hebb's work. It was not until the mid 1950s, when with the collaboration from Hebb and others, successful adaptations were made. Theories began to be modified.

In the summer of 1956, the Dartmouth Summer Research Project on Artificial Intelligence (AI) gave birth to neural nets. The following year, John von Neumann

suggested imitating simple neuron functions by using telegraph relays and vacuum tubes [17]. In 1958, a milestone paper by Frank Rosenblatt referred to the perceptron as a neural network structure [18]. The perceptron was simulated in detail on an IBM 704 computer at the Cornell Aeronautical Laboratory. The perceptron was probably the first 'learning machine'. The primary learning mechanism of a perceptron is 'self-organizing' or 'self-associative' in the sense that an initially random process tends to become organized.

In 1959, Bernard Widrow and Marcian Hoff developed models for ADALINE, and MADALINE (Multiple ADAptive LINear Elements). This was the first neural network applied to a real-world problem, *i.e.*, adaptive filters to eliminate echoes on phone lines. Widrow and Hoff's paper [19] is considered to be 'prophetic'. They suggested several practical implementations of their ADALINE:

> *" If a computer were built of adaptive neurons, details of structures could be imparted by the designer by training (showing it examples what he would like to do) rather than by direct designing. "*

Then came the so called 'Dark Age' for neural network research. In 1969, Marvin Minsky and Seymour Papert published a book [20], which condemned the Rosenblatt's perceptron. The charge was that it could not solve any 'interesting' problems. This brought a halt to much of the funding for neural network research. Many researchers turned their attention to expert systems, which make decisions based on a set of pre-defined rules under certain conditions.

Some of the disappointed researchers continued working. These included James Anderson, a neurophysiologist at Brown University, who developed a network model with the name of Brain-State-in-a-Box (BSB) [21]. Research also continued in Japan and Europe. Kunihiko Fukushima developed the Neocognitron, a neural network model for visual pattern recognition [22]. Teuvo Kohonen, an electrical engineer at Helsinki University, independently developed a similar model to Anderson's BSB [23]. There were others, plugging away quietly in their labs: Grossberg, Rumelhart and McClelland, Marr and Poggio, Amari, Cooper, and many others.

In 1982, John J. Hopfield at California Institute of Technology, published a paper reviving the neural network's field [24]. His follow-on paper was published in 1984 [25]. With clarity and with mathematical analysis, he showed how neural networks could work and what they could do. The future of artificial intelligence started to brighten once again. What had been thought to be the failures of artificial intelligence could now be overcome.

In 1985, the American Institute of Physics began an annual Neural Networks for Computing meeting. In 1987, the Institute of Electrical and Electronics Engineers' (IEEE) First International Conference on Neural Networks drew more than 1,800 attendees and 19 vendors. Later the same year, the International Neural Network Society (INNS) was formed under the leadership of Grossberg in U.S., Kohonen in Finland, and Amari in Japan.

The International Joint Conference on Neural Networks (IJCNN), held in June 1989, produced 430 papers, 63 of which focused on the application development. The

January 1990 IJCNN in Washington, D.C. included a concert of music generated by neural networks.

Today, there are numerous journals/magazines published monthly/quarterly on neural networks all over the world. A number of different conferences and symposia are held periodically and a large number of papers are presented by researchers on the various aspects and applications of the neural networks.

2.3 Artificial Neural Networks

This section presents a brief review of biological and artificial neurons followed by the network architecture. Several learning rules are discussed. Based upon these rules, different types of ANN models are introduced with a detailed mathematical treatment for the multilayer feedforward perceptron using backpropagation rule: the model subsequently used in the simulation of Digital Multilayer Neural Networks (DMNN).

2.3.1 Biological and Artificial Neurons

2.3.1.1 Biological Neuron

Before we describe the building blocks and operation of an artificial neural network, it is instructive to briefly examine the corresponding components of the brain which inspired neural computing. The basic anatomical unit in the biological nervous system is a specialized cell called the neuron. The brain is composed of about 10^{11} neurons of many different types. Figure 2.1 portrays a conceptual diagram of a typical

neuron. Most neurons can be divided into four distinct regions, each performing a specialized function. A tree-like network of nerve fiber called dendrites are connected to the cell body or soma, where the cell is located.

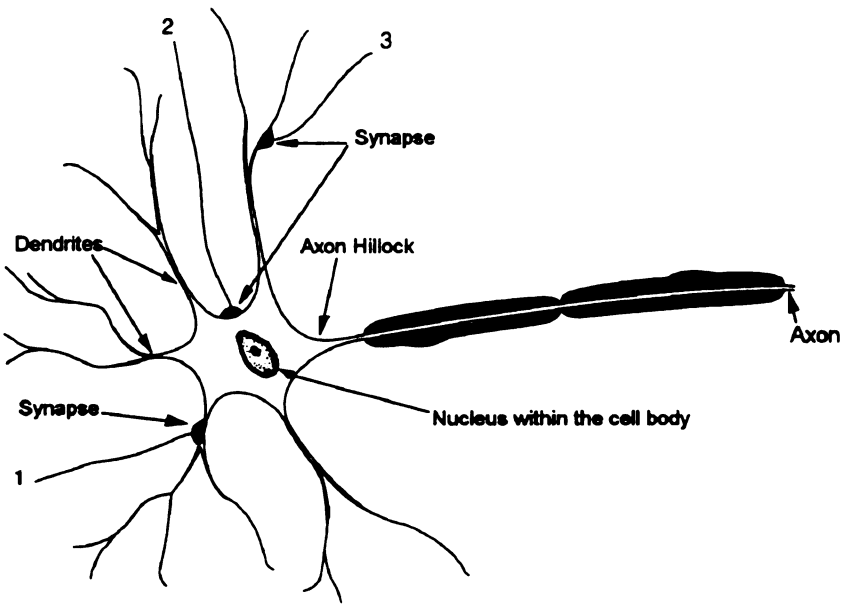


Figure 2.1. Biological Neuron [26].

Extending from the cell body is a single long fiber called the axon. The axon eventually branches or arborizes into strands and substrands. At the end of these are the transmitting ends of the synaptic junctions or synapses, to the other neurons. Receiving ends of these junctions on other cells can be found both on the dendrites (1&3 in Figure 2.1) and on the cell bodies themselves (2 in Figure 2.1). The axon of a typical neuron makes a few thousand synapses with other neurons [26].

are unidirectional in nature and determine the direction of signal flow. Axons are specialized for carrying signal towards other cells without much affecting its magnitude. Action potentials originate at the axon hillock, travel towards synapses and pass to other cells. Dendrites receive signals from sensory organs or from the axons of other neurons, convert these signals into electrical impulses and transmit them to the cell body. The cell body receives signals independently either through dendrites or through synapses directly connected with it. If electrical impulses are greater than a certain threshold, action potentials in the form of the pulse streams, are generated and actively conducted down the axon. The signal flow goes from dendrites, through the cell body and out through the axon. The axon terminal from a presynaptic cell sends chemical or electrical signals through a synaptic gap. The signals are collected by a postsynaptic cell.

Two types of synapses are believed to exist in a biological neural system: electrical synapses and chemical synapses. The two differ in both structure and function. Figure 2.2 shows the schematic of an electrical synapse. Cells communicating by electrical synapses are connected by gap junctions which allow an electrical pulse to pass from the presynaptic cell to the postsynaptic cell. Action potential is generated in the postsynaptic cell. Chemical substances, called neurotransmitters, are responsible for transmitting the signals in chemical synapses [27].

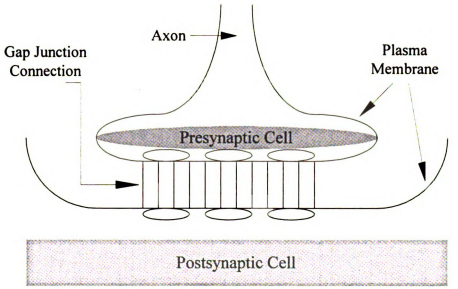


Figure 2.2. Electrical Synapse [1].

Two types of signals are generated in synapses: excitatory and inhibitory. In excitatory synapse, the signal from the presynaptic cell causes a change in the plasma membrane of the post synaptic cell that tends to induce an action potential. Action potentials are pulse streams with a pulse-width of about 1 msec. With an inhibitory synapse, a nerve impulse in a presynaptic neuron prevents the generation of an action potential by affecting the electrical properties of the post synaptic membrane. Excitatory and inhibitory stimuli often affect a single neuron in combination [1].

2.3.1.2 Artificial Neuron

An artificial neuron, a unit analogous to a biological neuron is referred to as a processing element (PE). Just as a biological neuron, the artificial neuron is a many-input

single-output processing element. Each input is weighted. The output is a linear or nonlinear function of weighted sum of its inputs. Figure 2.3 portrays a single artificial neuron.

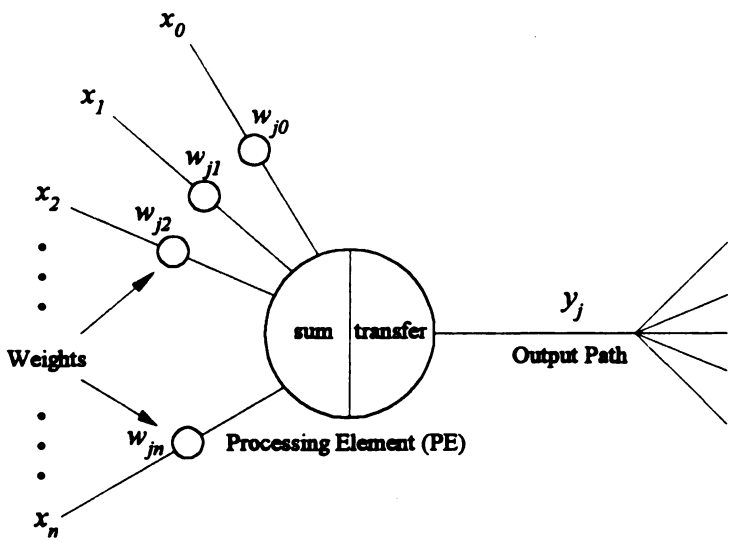


Figure 2.3. Artificial Neuron.

The inputs and the weights on the inputs are represented as vectors, $[x_1, x_2, ..., x_n]$ and $[w_1, w_2, ..., w_n]$, respectively. The total input signal is the dot or inner product of the two vectors and this is compared to a threshold value to determine the output. If the total input is greater than the threshold value, the processing element fires (excitatory), and if it is less, the processing element does not fire (inhibitory). Mathematically, if y is the output of a neuron and θ is the threshold value, the required neuron transfer function can be expressed as

$$y = f\left(\sum_{j=1}^{n} x_j w_j - \theta\right) \tag{2.1}$$

where $f(\cdot)$ is a linear or nonlinear threshold function. Some of the commonly used nonlinear functions are shown in Figure 2.4. The most pervasive threshold function is the sigmoid function because it is a bounded, monotonic and non-decreasing function that provides a graded, nonlinear response, most resembling a biological neuron.

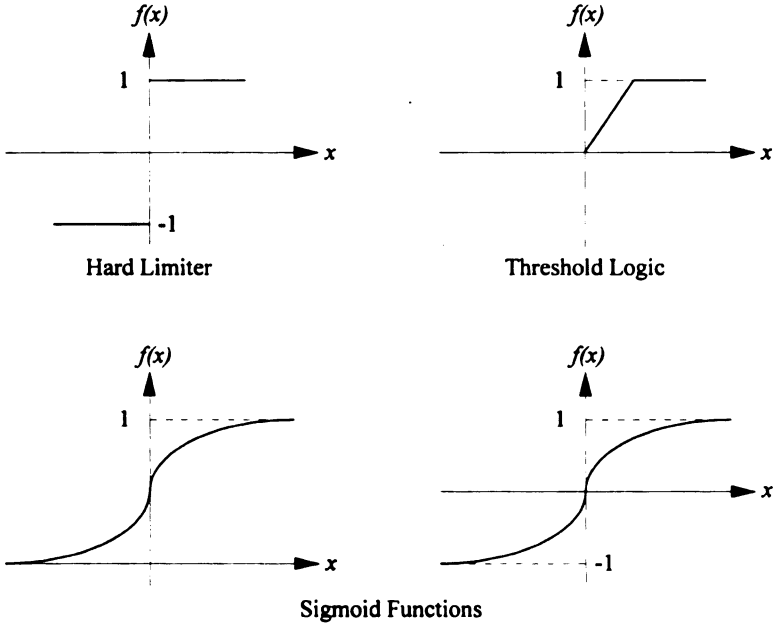


Figure 2.4. Sample Activation Functions.

2.3.2 Network Architecture

So far a single neuron (processing element) has been discussed. These processing elements are combined together to form a layer of neurons whose cell body is called a

node. Inputs could be connected to many nodes with different weights. Each node has an individual output. A neural network layer is shown in Figure 2.5.

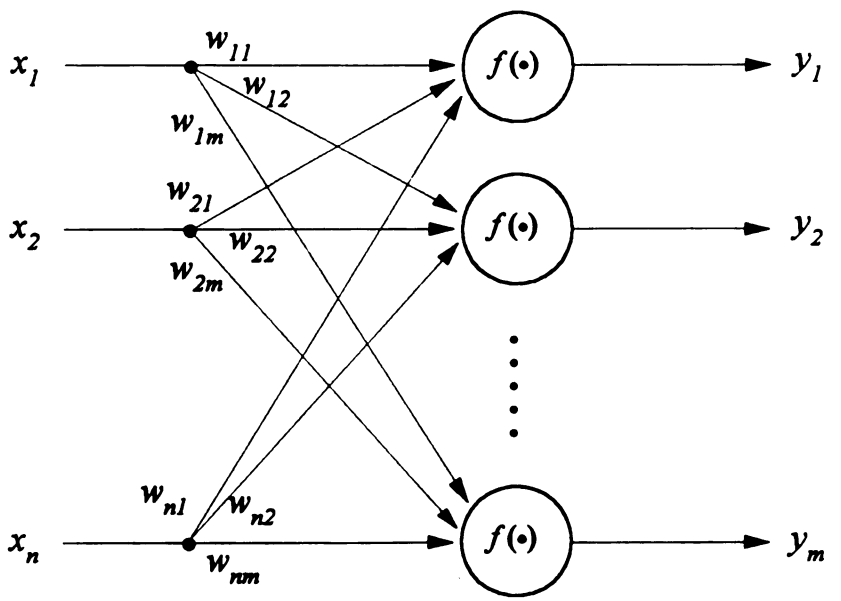


Figure 2.5. Neural Network Layer.

Several layers can be interconnected to form a neural network. The layer that receives the inputs is called the input layer or input buffer. The network output is generated from the output layer or the output buffer. The intermediate layers are called the hidden layers. The network is said to be fully connected if every output from one layer is passed along to every node in the next layer. Figure 2.6 shows a fully connected network.

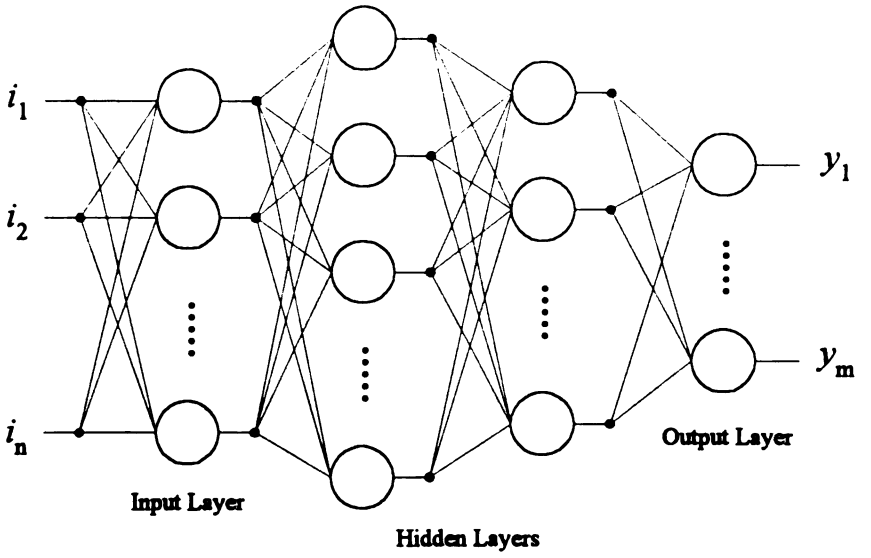


Figure 2.6. A Fully Connected Neural Network.

2.3.3 Learning Rules

The main characteristic of a neural network is its ability to learn. Artificial neural networks are constructed according to some learning rule used to change the interconnected weights. In a very broad sense, neural networks employ two modes of operation. In the first mode, the operational mode, a neural network computes the output for a given set of inputs. This mode assumes that the interconnected weights are fixed to some desired values. The other mode, called the learning mode, the interconnected weights are adjusted using an appropriate learning law with the objective of training the neural network to give a desired output. In most cases, the operational mode is followed by the learning mode, that is, first the desired accuracy is achieved through the learning process and then the neural network is used in the operational mode.

The weights can be dynamically adjusted to produce a given output. This dynamic modification of the weights is the very essence of the learning process. Learning is the process by which a neural network modifies its weights in response to the external inputs. The equation that specifies this change is called the learning law, or the learning rule. Before discussing various learning rules, learning types, that is, supervised, unsupervised, and reinforced learning are defined.

Supervised learning requires a teacher. The teacher may be a training data set or an observer who grades the performance. In either case, the teacher is used to form an error signal which is used in some kind of law to update the interconnected weights of the neural network. When there is no external teacher, the neural net must organize itself by utilizing some internal criteria designed into the network. This type of learning, called unsupervised learning is based on the input/output values of the neural net. This kind of learning is often used in the feature extraction applications. For reinforced learning, the weights connected to a neuron are not updated in direct or inverse proportion to the error signal of that particular neuron, but instead they are updated in proportion to some type of global reinforcement signal. A binary or a trinary neural network is an example of such type of learning. In a binary neural net, weight update can only take two values (+1 or -1) whereas in a trinary network three update values are allowed (+1, 0, -1).

Many learning laws are in the common use. Research has continued, however, and new ideas are being tried. The definition of a few learning laws, generally omitting the mathematical details, is given here.

2.3.3.1 Hebb's Rule

The first rule was introduced by Donald Hebb. He stated in his book [16]:

" If a processing element receives an input from another processing element, and if both are highly active (mathematically have the same sign), the weight between the processing elements should be strengthened. "

2.3.3.2 The Delta Rule

This rule is also known as the Widrow-Hoff Learning Rule. The Delta Rule is based on the idea of continuously modifying the strengths of the connections to reduce the difference (delta) between the desired output value and the current output value of the processing element. If there is no difference between the two, no learning takes place. The rule for changing the weights, following the presentation of the input/output pair p, in case of no hidden layers, is given by

$$\Delta_p w_{ij} = \eta \left(t_{pj} - o_{pj} \right) i_{pi} = \eta \, \delta_{pj} \, i_{pi}, \qquad (2.2)$$

where t_{pj} is the target input for the jth component of the output pattern for pattern p, o_{pj} is the jth element of the actual output pattern produced by the presentation of input pattern p, i_{pi} is the value of the ith element of the input pattern, $\delta_{pj} = t_{pj} - o_{pj}$, and $\Delta_p w_{ij}$ is the change to be made to the weight from the ith to the jth unit following the presentation of the pattern p. Widrow and Hoff used this rule in their ADALINE model [28].

2.3.3.3 Gradient Descent Rule

In this rule the weights are modified by an amount proportional to the first derivative of the error with respect to the weight. If w_{ij} is the connection weight between the neuron i and neuron j, and E_p is the error cost function for the pattern p; the gradient descent rule states

$$\Delta_p w_{ij} \propto -\frac{\partial E_p}{\partial w_{ij}} . \qquad (2.3)$$

This corresponds to performing the steepest descent on a surface in the weight space whose height at any point in the weight space is equal to the error measure [3]. Due to a mathematical approach to minimize the error between the actual and the desired outputs, the gradient descent rule converges to a point of stability very slowly.

2.3.3.4 Kohonen's Learning Law [2,29]

This rule is based on the unsupervised learning approach. In this procedure, the processing elements compete for the opportunity of learning. The processing element with the largest output is declared the winner and has the capability of inhibiting its competitors as well as exciting its neighbors. Only the winner is permitted an output, and only the winner plus its neighbors are permitted to adjust their weights. Further, the size of the neighborhood can vary during the training period. The usual pattern is to start with a larger definition of the neighborhood, and narrow it as the training proceeds.

2.3.3.5 Grossberg Learning Rule [2]

In the Grossberg's model, every neural network is made up of the instars and the outstars. An instar is a processing element receiving many inputs and an outstar is a processing element sending its output to many other processing elements. The connections allow recall of a concentrated image from a single outstar node. The pattern is stored distributively. If both the input and the output activities of a node are high, the weights change significantly. If either the total input or the output is small, the change in the weights is very small, and the weights could even approach to 0 on unimportant connections. Time is important in Grossberg learning. If an input stimulus is removed over time, the output response reduces as forgetting sets in. The network threshold is important: if set too high, the network responds to every little nuance and if it is set too low, the network ignores too much. Learning is turned off during any recall.

2.3.3.6 Backpropagation Learning Rule

The backpropagation of errors technique is the most commonly used generalization of the Delta Rule. This technique involves two phases. The first phase, called the forward phase, occurs when the input is presented and propagated forward through the network to compute an output value for each processing element. For each processing element, all the current outputs are compared with the desired output, and the difference, or the error, is computed. In the second phase, called the backward phase, the recurring difference computation (from the first phase) is performed in a backward direction. The new inputs can only be presented when these two phases are complete.

Generally this technique is applied to the hierarchical networks involving hidden layers. At the output layer, the actual output from each node and the expected output is known. The 'magic' of this rule is, how the adjustment of weights take place in hidden layers [2].

There are several important drawbacks: backpropagation is very slow, requires much off-line training, exhibits temporal instability (can oscillate), and has the tendency to get stuck at a local minima (see Section 4.3).

2.3.4 Neural Network Models

There are basically three types of neural network models: feedback models, feedforward models and recurrent models. A brief description of these models is presented here. A detailed mathematical treatment has been given to the multilayer feedforward perceptron model to facilitate its subsequent use.

2.3.4.1 Feedback Model

In feedback neural networks, the neural elements are connected to one another by the feedback paths from outputs to the inputs of the neural elements. A key issue for these networks is the definition of an energy function which always decreases during the dynamical evolution. The Hopfield model is a typical example of the feedback models.

The Hopfield model is a one-layered feedback network which consists of interconnected nonlinear neurons. Many implementations have been built using this model. One version of the original net [2,24] which can be used as a content addressable

memory is described here. This net, shown in Figure 2.7, has N nodes containing hard limiting nonlinearities and the binary inputs and the outputs taking on the values +1 and -1. The output of each node is fed back to all the other nodes via the weights denoted w_{ij}.

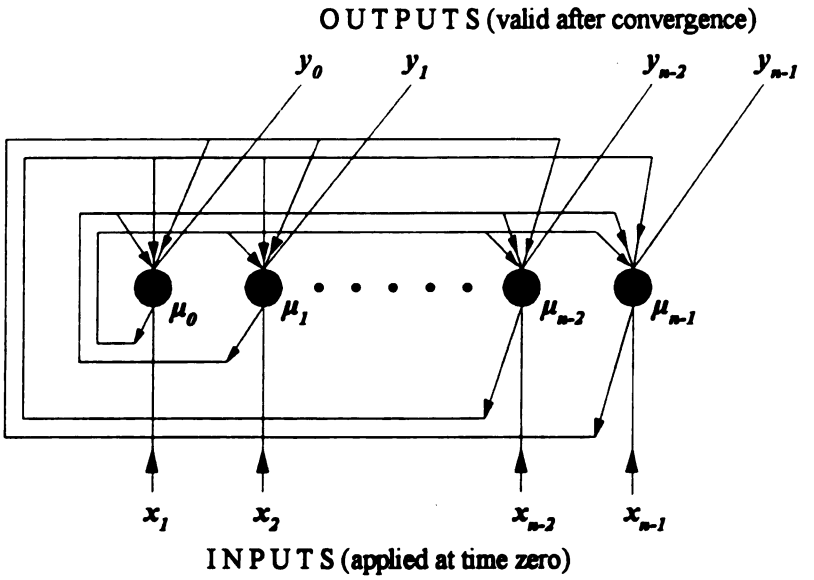


Figure 2.7. Hopfield Neural Network Model [2].

The weights are first set from exemplar patterns for all classes using

$$w_{ij} = \begin{cases} \sum_{s=0}^{M-1} x_i^s x_j^s, \cdots\cdots & i \neq j \\ 0, \cdots\cdots\cdots\cdots & i = j,\ 0 \leq i, j \leq M-1, \end{cases} \tag{2.4}$$

where w_{ij} is the connection weight from node i to node j. x_i^s, which can be +1 or -1, is the element i of the exemplar for class s and M is the total number of classes.

An unknown pattern is imposed on the net at time zero by forcing the output of the net to match the unknown pattern, *i.e.*,

$$\mu_i(0) = x_i, \cdots\cdots\cdots 0 \le i \le N-1, \tag{2.5}$$

where μ_i (t) is the output of node i at time t and x_i, which can be +1 or -1, is element i of the input pattern. N is the total number of nodes.

Following this initialization, the net iterates in descrete time steps expressed using the formula

$$\mu_j(t+1) = f_h\left[\sum_{i=0}^{N-1} w_{ij}\,\mu_i(t)\right], \cdots\cdots 0 \le j \le M-1, \tag{2.6}$$

where f_h is the hard limiting nonlinearity. The process is repeated until node outputs become constant on successive iterations. The node outputs then represent the exemplar pattern that best matches the unknown input. The pattern specified by the node outputs, after convergence, is considered the net output.

Hopfield has proven that the net converges when the weights are symmetric (w_{ij}=w_{ji}) and the node outputs are updated asynchronously using the equations mentioned above [24]. Hopfield also demonstrated that the net converges when the graded nonlinearity, similar to the sigmoid nonlinearity in Figure 2.4, is used [25]. The Hopfield model has also been applied to the combinatorial optimization problems where it converges to a good solution in a few time constants [30,31]. When the Hopfield network is used as an associative memory, the net output after convergence is used directly as the complete restored memory. While applying the Hopfield model as a classifier, the output after convergence has to be compared to the M exemplars to determine if it matches an exemplar exactly. If it does, the output is that class whose exemplar matches the output pattern. If it does not, then a 'no match' result occurs [2].

2.3.4.2 Feedforward Model

Feedforward networks were first studied in detail by Rosenblatt in the early 1960's [32]. A number of algorithms for training of these networks have been developed since then. These networks can learn a set of input-output pairs as examples and can successfully generalize the learned patterns. Feedforward networks have been applied to pattern recognition, robotics, and control problems [33,34].

2.3.4.2.1 Single Layer Perceptron

This net generated much interest when developed initially because of its ability to learn to recognize simple patterns. A simple perceptron is a single layered feedforward neural network consisting of n inputs and an output layer. Figure 2.5 illustrates an example of a single layer perceptron. x_i^{μ} is the ith element of the input pattern and y_i^{μ} is the output of the neuron i when pattern μ is presented to the network. w_{ij} is the connection weight between the neuron i and the jth element of the input pattern. If the number of patterns is p such that $\mu = 1,2,...,p$, the output of the output layer can be described by

$$y_i^{\mu} = f\left(\sum_{j=1}^{n} w_{ij}\, x_j^{\mu} + \theta_i\right) \tag{2.7}$$

$$y_i^{\mu} = f\left(\sum_{j=0}^{n} w_{ij}\, x_j^{\mu}\right), \tag{2.8}$$

where $x_0^{\mu}=1$ for all μ and $w_{i0}=\theta_i$ is a bias. $f(\cdot)$ is the continuous, nondecreasing and differentiable function.

The system's performance is measured by the error cost function which is defined as

$$\begin{aligned} E &= \frac{1}{2}\sum_{\mu=1}^{p} E^{\mu} \\ &= \frac{1}{2}\sum_{i}\sum_{\mu}\left(t_i^{\mu} - y_i^{\mu}\right)^2 \\ &= \frac{1}{2}\sum_{i}\sum_{\mu}\left[t_i^{\mu} - f\left(\sum_{j} w_{ij} x_j^{\mu}\right)\right]^2, \end{aligned} \tag{2.9}$$

where t_i^{μ} is the desired output of the neuron i for the input pattern μ. The connection weights, w_{ij}, are changed by the gradient descent rule (Section 2.3.3.3)

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{2.10}$$

$$= \eta \sum_{\mu=1}^{p}\left(t_i^{\mu} - y_i^{\mu}\right) f'\left(\sum_{j} w_{ij} x_j^{\mu}\right). \tag{2.11}$$

The simple perceptron can not solve problems in which input patterns are linearly dependent, and may offer alternate partial solutions. However, multilayer feedforward neural networks with nonlinear neuron elements can overcome this limitation [1].

2.3.4.2.2 Multilayer Perceptron

Multilayer perceptrons are backpropagation neural networks consisting of an input layer, an output layer and one or more hidden layers of neurons. Figure 2.8 depicts a neural network which has one input layer, I, one output layer, K, and one hidden layer, J. The input layer, I, consists of n_i neurons, the output layer, K, consists of n_k neurons, and the hidden layer, J, consists of n_j neurons.

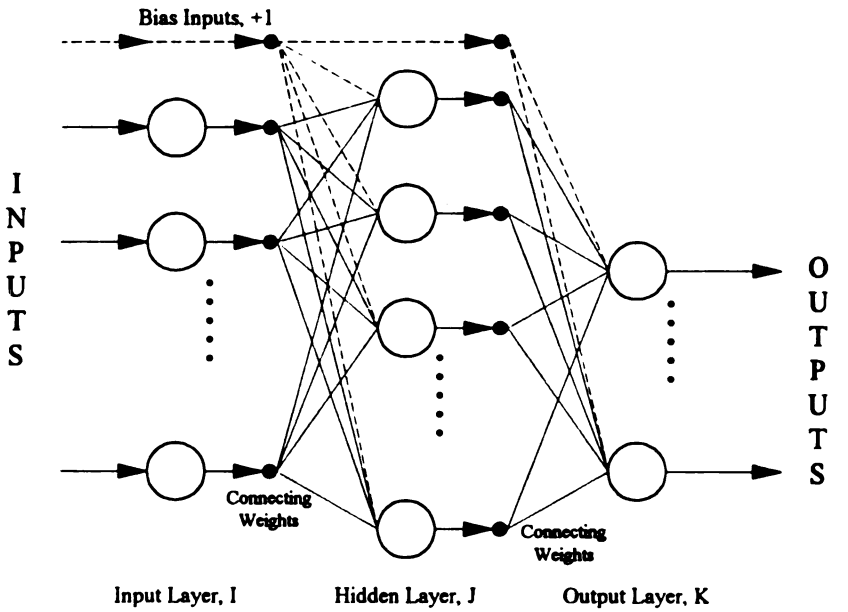


Figure 2.8. A Three Layer Neural Network.

The mathematical expressions for the network are

$$O_k = f\left(\sum_{j=0}^{n_j} H_j W_{jk}\right) \quad ; \quad H_j = 1.0 \text{ when } j = 0 \tag{2.12}$$

and $$H_j = f(\sum_{i=0}^{n_i} I_i W_{ij}) \quad ; \quad I_i = 1.0 \text{ when } i = 0, \tag{2.13}$$

where $f(\cdot)$ is a continuous differentiable function, W_{jk} is the adjustable weight parameter connecting the jth hidden layer neuron to the kth output layer neuron, and W_{ij} is the adjustable weight parameter connecting the ith input layer neuron to the jth hidden layer neuron.

The outputs, O_k for the output layer K and H_j for the hidden layer J, are calculated using equations (2.12) and (2.13), for $k = 1,2,...,n_k$ and $j = 1,2,...,n_j$, respectively. Y_k^d are the desired outputs for $k = 1,2,...,n_k$. The objective is to minimize the difference between the desired outputs, Y_k^d, and the observed outputs, O_k, for $k = 1,2,...,n_k$. Conventionally, the error cost function for each k is defined as

$$e_k = \frac{1}{2}(Y_k^d - O_k)^2 \quad \text{for} \quad k = 1,2, \ldots ,n_k \; . \tag{2.14}$$

The variable parameters in the neural networks are the connecting weights that can be adjusted by minimizing the cost function. Therefore, minimizing the cost function for the kth output node with respect to the weight connecting jth hidden node, produces

$$\frac{\partial e_k}{\partial W_{jk}} = -(Y_k^d - O_k)\frac{\partial O_k}{\partial W_{jk}} \; . \tag{2.15}$$

Substituting equation (2.12) in equation (2.15) yields

$$\frac{\partial e_k}{\partial W_{jk}} = -(Y_k^d - O_k) f'(\sum_{j=0}^{n_j} H_j W_{jk}) H_j \quad ; \quad H_j = 1.0 \text{ when } j = 0 \; . \tag{2.16}$$

The *k*th gradient of the error cost function, equation (2.15), with respect to the weight parameter connecting the *j*th hidden neuron to the *k*th output neuron is given by equation (2.16). Defining

$$\delta_k = (Y_k^d - O_k) f'(\sum_{j=0}^{n_J} H_j W_{jk}) \tag{2.17}$$

and substituting the value of δ_k in equation (2.16) leads to

$$\frac{\partial e_k}{\partial W_{jk}} = -\delta_k H_j . \tag{2.18}$$

Backpropagation is a gradient descent learning rule that minimizes the error cost function, equation (2.14), according to the following formula

$$W_{jk}[n+1] = W_{jk}[n] + \eta(-\frac{\partial e_k}{\partial W_{jk}}) + \alpha \Delta W_{jk}[n]. \tag{2.19}$$

Substituting equation (2.18) in (2.19), gives the backpropagation law to update the weight parameters connecting the hidden layer to the output layer. The expression is given as

$$W_{jk}[n+1] = W_{jk}[n] + \eta \delta_k H_j + \alpha \Delta W_{jk}[n] \tag{2.20}$$

where n is the time index, η is the learning rate, α is the momentum factor, and $\Delta W_{jk}[n]$ is the change in weight, connecting the *j*th hidden neuron to *k*th output neuron, in the last step.

Equation (2.20) can be easily implemented for weights connecting the output layer to the last hidden layer, for example, in Figure 2.8 for layer K to layer J. However, equation (2.20) can not be used to update weights connecting any other two layers. For

example, weights, W_{ij}, connecting the input layer, I, to the hidden layer, J, can not be adjusted using equation (2.20). The reason being that there is no known target response for the hidden layer neurons. However, the contribution in the total error by the hidden layer neurons at the output layer neurons must be propagated back. In Figure 2.8, weights W_{ij} are updated reflecting back the error from all the output nodes to the jth hidden node. Mathematically, this is accomplished by the relationship

$$W_{ij}[n+1] = W_{ij}[n] + \eta(-\sum_{k=0}^{n_k}\frac{\partial e_k}{\partial W_{ij}}) + \alpha \Delta W_{ij}[n]. \tag{2.21}$$

In equation (2.21) the factor involving partial derivatives is calculated using the chain rule given by

$$\begin{aligned}\sum_{k=0}^{n_k}\frac{\partial e_k}{\partial W_{ij}} &= \sum_{k=0}^{n_k}\frac{\partial}{\partial W_{ij}}[\frac{1}{2}(Y_k^d - O_k)^2] \\ &= \sum_{k=0}^{n_k}\frac{\partial e_k}{\partial O_k}\bullet\frac{\partial O_k}{\partial H_j}\bullet\frac{\partial H_j}{\partial W_{ij}} \\ &= -\sum_{k=0}^{n_k}(Y_k^d - O_k)\bullet f'(\sum_{j=0}^{n_j}H_j W_{jk})W_{jk}\bullet\frac{\partial H_j}{\partial W_{ij}}.\end{aligned} \tag{2.22}$$

Substituting equation (2.17) in equation (2.22) gives

$$\sum_{k=0}^{n_k}\frac{\partial e_k}{\partial W_{ij}} = -\frac{\partial H_j}{\partial W_{ij}}\bullet\sum_{k=0}^{n_k}\delta_k W_{jk}\ . \tag{2.23}$$

From equation (2.13), we have

$$\frac{\partial H_j}{\partial W_{ij}} = f'(\sum_{i=0}^{n_i}I_i W_{ij})I_i\ . \tag{2.24}$$

Substitution of equation (2.24) in equation (2.23) gives

$$\sum_{k=0}^{n_k} \frac{\partial e_k}{\partial W_{ij}} = -f'(\sum_{i=0}^{n_i} I_i W_{ij}) I_i \bullet \sum_{k=0}^{n_k} \delta_k W_{jk} \ . \tag{2.25}$$

Then, letting

$$\delta_j = f'(\sum_{i=0}^{n_i} I_i W_{ij}) \sum_{k=0}^{n_k} \delta_k W_{jk} \tag{2.26}$$

allows equation (2.25) to be written as

$$\sum_{k=0}^{n_k} \frac{\partial e_k}{\partial W_{ij}} = -\delta_j I_i \ . \tag{2.27}$$

Finally, substituting equation (2.27) into equation (2.21) produces

$$W_{ij}[n+1] = W_{ij}[n] + \eta \delta_j I_i + \alpha \Delta W_{ij}[n] \ . \tag{2.28}$$

Similarities between equations (2.20) and (2.28) are obvious. In a more general form, the parameter adjustment between a node a and a node b can be given as

$$W_{ab}[n+1] = W_{ab}[n] + \eta \delta_b S_a + \alpha \Delta W_{ab}[n] \ . \tag{2.29}$$

Here S_a, the symbolizing source, is the output of node a or an external input to the network and δ_b is an associated error at the node b. δ_b can be calculated as

$$\delta_b = S_b'(Y_b^d - S_b) \tag{2.30}$$

for the output layer nodes, and

$$\delta_b = S_b' \sum_b W_{bc} \delta_c \tag{2.31}$$

for the hidden layer nodes.

2.3.4.3 Recurrent Model

In recurrent networks, connections in either direction, *i.e.*, between a pair of layers and within a layer to itself, are possible. The Boltzmann machine is a typical example of the recurrent networks with symmetric connections.

The Boltzmann Machine is comprised of visible units (input units and output units) and hidden units [35,36]. Figure 2.9 illustrates the structure of a Boltzmann machine. The units are stochastic and take output value $v_i = \pm 1$ with a certain probability. A simulated annealing procedure is used to achieve a global minimum. Boltzmann machines have been used for pattern recognition and the optimization problems. However, these networks take a very long time to converge and their hardware implementation is not practical.

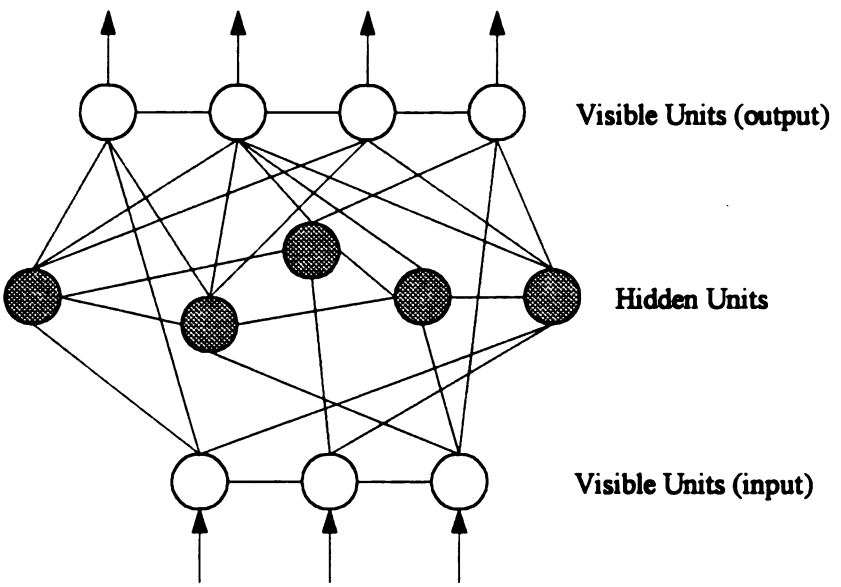


Figure 2.9. Boltzmann Machine.

2.4 Neural Network Implementation

The preliminary efforts of neural network modeling were mostly theoretical since the implementation tools had not been developed with the same pace. As sophistication in the field of computers progressed, many practical models have been presented. Software simulations have not only moved this fledgling technology along, but also provided an environment that nurtures testing and experimentation. The goal of many researchers has been, and still is, to find implementations which are fast and yet not too expensive. This generally means implementing in hardware, however, many current models have been simulated in software using serial or parallel digital computers. To date a number of hardware and software prototypes have been built using electronic/opto-electronic techniques and employing various programming tools. Network implementations can be divided into two broad categories: hardware implementation and software implementation. This section briefly discusses these categories.

2.4.1 Hardware Implementation

Hardware implementation can be divided into two categories based upon the method used to express the values within the network: analog, and digital.

2.4.1.1 Analog Implementation

Analog implementations are mostly based upon the physical properties of analog components such as the behavior of voltage/current in resistors, capacitors, and semiconductor devices. The interconnections in an analog neural network can be simple fixed value resistors representing the synaptic weight values. Such networks, however, are not of much practical significance since these networks can not be made programmable due to constant value synaptic weights.

To design a programmable network, the synaptic weights have to be stored in a memory. A weight can be stored as the voltage difference between the two capacitors [37]. This technique, however, requires a refresh circuitry to overcome the problem of leakage in the capacitors. Static memory cells have been used for storage of weight values [6]. This method is also not very efficient since most applications require a higher resolution. The weights can also be stored digitally but this technique requires a D/A converter at each connection to perform analog multiplication of the stored weights with the input signal.

Hardware implementations using analog computations are less expensive because the network operations can be performed using inexpensive hardware. This category however, suffers from a number of disadvantages, such as low accuracy and limited dynamic range, due to physical constraints of the analog components. The design flexibility in analog implementation is also strictly constrained because only mathematical functions resulting from physical principles are available.

2.4.1.2 Digital Implementation

Since the artificial neural networks require a large amount of interconnections and computation, especially in training, the development of a specialized high performance processor is needed. Digital implementation is a reasonable alternative for realizing neural network VLSI processor. Several digital neural networks based on custom VLSI design have been developed where a neuron is a processing element consisting of computing units, registers, and a lookup table. Suzuki and Atlas mapped a network to an array of custom processors [38,39]. This ANN hardware has a high design flexibility, but suffers from a large hardware requirement.

In order to achieve a high design flexibility and high chip density, digital networks using stochastic computing techniques have been proposed [9,10,13]. In this approach algebraic operations are replaced by stochastic processes using pseudo-random pulse sequences. Simple logic gates alongwith other digital components perform multiplication and nonlinear transformation of signals. The values for synaptic weights and the input operands are restricted in the range between 0.0 and 1.0 both for training and testing [1]. An operand x in the pulse mode representation is the probability of a pulse occurrence in the corresponding binary sequence $x_{(n)}$ at each clock. Stochastic computations using random pulse sequences inherently utilize concurrent processing in all synaptic and neuron elements. Use of simple logic gates allows high neuron density and compact network architecture. The network can be made programmable thus increasing the design flexibility. A pulse mode digital multilayer neural network has been developed by Kim [1]. Details of Kim's model are presented in the next chapter.

2.4.2 Software Implementation

Software implementation or simulations can be affected either on general purpose computers or on special purpose computers. General purpose parallel computers with a large number of processing elements are being used for ANN simulations. Processing elements communicate through a single high speed data path and may have a dedicated memory to store data.

Warp machine was used to implement a backpropagation network using a systolic array of 10 processing elements.[40]. Forrest, *et al.*, used a Distributed Array Processor (DAP) comprised of 4096 processors to implement a Hopfield network [41]. The DAP was able to perform 25 million additions per second. General purpose parallel computers can solve problems of limited size within a reasonable time. But, as the size of ANN increases, the computational load increases beyond the acceptable bounds and these computers take hours to solve a given problem.

Special purpose computers designed for ANN simulations are called neurocomputers. Neurocomputers are attached to a host computer as coprocessors and are controlled through a user program. Mark III and Mark IV neurocomputers were developed by TRW [42]. These machines were able to process up to 450,000 and 5,000,000 interconnections per second respectively.

CHAPTER 3

DIGITAL MULTILAYER NEURAL NETWORKS

3.1 Introduction

Artificial Neural Networks present a practical approach to solving many computationally intensive and ill-defined problems such as pattern recognition, optimization, and other complex information processing tasks. Kim [1] has developed a new architecture for a digital feedforward neural network. A statistical model of the digital multilayer neural network (DMNN) is also evolved based on stochastic computing. Kim presents a compact and expandable pulse mode DMNN architecture employing simple logic gates and modular design techniques. The colossal parallelism embedded in the stochastic computations, using random pulse streams, is thoroughly explored with this architecture. Kim makes use of VHDL hardware description language to model and simulate all the components of the DMNN coprocessor. Kim also demonstrates the applicability of his model by using benchmark and character recognition problems [1,43].

This chapter presents a brief overview of Kim's work with a view to develop a thorough understanding of the DMNN.

3.2 Stochastic Computing in DMNN

3.2.1 Generating Probability

Stochastic computing techniques, using random pulse streams, were proposed in the 1960s [44,45]. In stochastic computations, the operands are normalized and represented by probabilities which are actually encoded in random pulse streams. Probability is estimated as a relative frequency of 1 pulse occurrences in a finite but long pulse stream. Since the probability can not be measured exactly, errors by estimation are introduced in the form of variance when the stochastic computing techniques are used. This idea has been used as an alternative to the deterministic computations in the area of artificial neural networks since late 1980s. The main reason is that stochastic computing using random pulse sequences shares important characteristics with ANN dynamics, *i.e.*, network performance depends on the collective properties of the network where each processing element does not necessarily perform correct computations.

Stochastic computing techniques require that values of all the operands lie between 0 and 1. The fractional numbers represented by the probabilities are encoded in the random pulse streams. If the fractional number is stored in an n-bit register, the resolution is $1/(2^n-1)$. The random pulse streams corresponding to a fractional number can be generated by comparing the number with a pseudo random number. The pseudo random number can be generated from a PN sequence by taking all the bits of a tapped Linear Feedback Shift Register (LFSR) in parallel. An n-bit LFSR is shown in Figure 3.1.

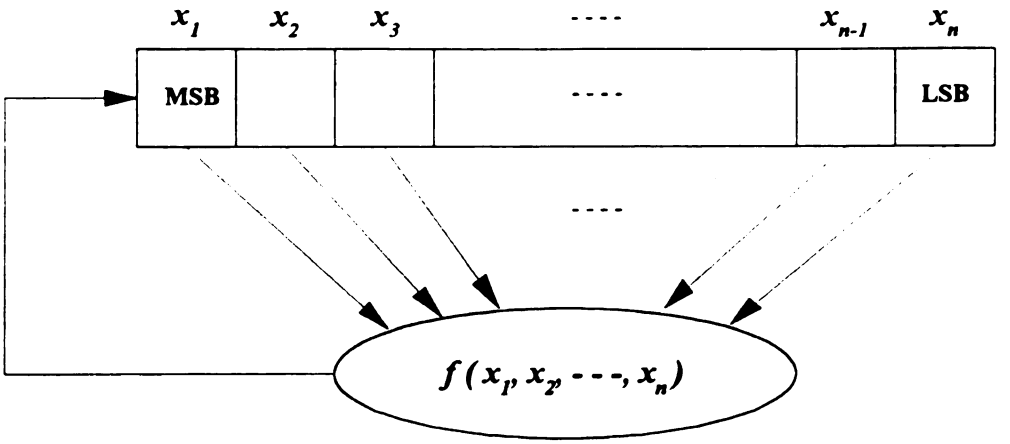


Figure 3.1. Block Diagram of a Linear Feedback Shift Register (LFSR) [1].

Fractional numbers from $1/(2^n-1)$ to 1, equally spaced by $1/(2^n-1)$, are generated exactly once in a period (2^n-1). The distribution of the pseudo random number is close to an ideal uniform distribution. Figure 3.2 shows the diagram of a Random Pulse Generator (RPG) for a fractional number x. The generating probability x is the probability of pulse occurrence in the corresponding random pulse sequences $x_{(n)}$ at each clock pulse. x is estimated in a sampling clock period. The sampling clock period is defined as finite clock periods taken for the estimation of x.

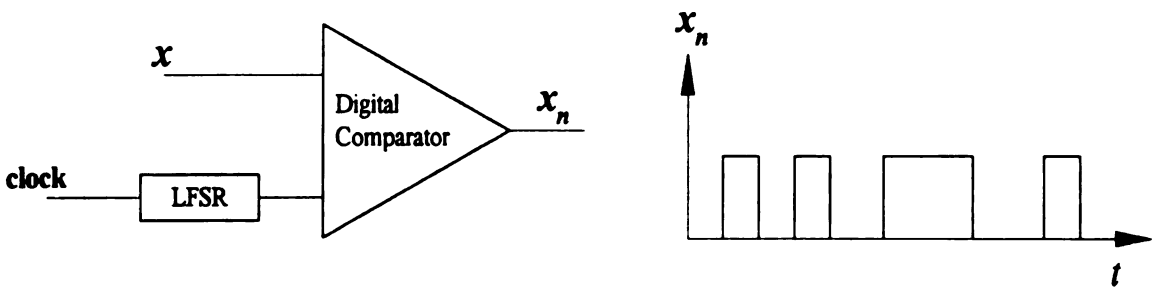


Figure 3.2. Random Pulse Generator (RPG) [1].

Error (noise) is involved in estimating the generating probability in finite clock periods. Estimate $\hat{x}$ is taken as the original signal plus random noise. The generating probability of the random pulse generator has been modeled as a binomial distribution in the literature [45]. However, the pulse occurrence in $x_{(n)}$ is not perfectly independent because a maximum length LFSR generates fractional numbers between $1/(2^n-1)$ and 1 such that each number occurs exactly once in a clock period. Accordingly, the pulse occurrence in $x_{(n)}$ has statistical dependency. The sampling distribution of x can be modeled closely by a hypergeometric distribution. The details of this statistical model are given in [1]. This new statistical model has been used to perform an analysis on random noise effects in digital multilayer neural networks (DMNN).

3.2.2 Basic Stochastic Computing

Stochastic computing exploits the similarities between probability algebra and Boolean algebra. A random pulse sequence $x_{(n)}$ is a sequence of pulses whose probability x can not be measured during any one clock period. It can, however, be approximated by a measurement of the average pulse rate. Any Boolean operation over individual pulses corresponds to an algebraic operation among variables represented by their respective average pulse rates [45]. If two sequences $x_{(n)}$ and $y_{(n)}$ are statistically independent, the probability of pulse occurrence in an output sequence $z_{(n)}$ of an AND gate is

$$z = P[z_{(n)} = 1] \tag{3.1}$$

$$= P[x_{(n)} = 1 \wedge y_{(n)} = 1] \tag{3.2}$$

$$= P[x_{(n)} = 1]\, P[y_{(n)} = 1] \tag{3.3}$$

$$= x\, y\,. \tag{3.4}$$

The probability of pulse occurrence in an output sequence $z_{(n)}$ of an OR gate is

$$z = P[z_{(n)} = 1] \tag{3.5}$$

$$= P[x_{(n)} = 1 \vee y_{(n)} = 1] \tag{3.6}$$

$$= P[x_{(n)} = 1] + P[y_{(n)} = 1] - P[x_{(n)} = 1 \wedge y_{(n)} = 1] \tag{3.7}$$

$$= x + y - x\, y\,. \tag{3.8}$$

Instead of being statistically independent, if the two sequences are mutually exclusive, *i.e.*, $x\, y = 0$, the logical OR performs a direct summation.

A NOT gate produces an output pulse whenever no input pulse occurs. If $x_{(n)}$ is the input sequence of a NOT gate, the probability of pulse occurrence in the output sequence $z_{(n)}$ is

$$z = P[z_{(n)} = 1] \tag{3.9}$$

$$= 1 - P[x_{(n)} = 1] \tag{3.10}$$

$$= 1 - x\,. \tag{3.11}$$

Examples of stochastic computations utilizing the duality between Boolean and algebraic operations can be found in [45].

3.2.3 Stochastic Computing in the DMNN

Neural operations in the DMNN are performed with basic gates using pulse sequences as inputs. Let w_{ij} and v_j be the connection weight between neurons i and j, and the neural activation of neuron j, respectively. If two sequences $w_{ij(n)}$ and $v_{j(n)}$ are statistically independent, the probability of pulse occurrence in an output sequence $m_{ij(n)}$ of an AND gate is

$$m_{ij} = P[m_{ij(n)} = 1] \tag{3.12}$$

$$= P[w_{ij(n)} = 1 \wedge v_{j(n)} = 1] \tag{3.13}$$

$$= P[w_{ij(n)} = 1] P[v_{j(n)} = 1] \tag{3.14}$$

$$= w_{ij} v_j . \tag{3.15}$$

Input summation and nonlinear transformation is performed simultaneously using a logical OR operation. The inputs of an OR gate are product sequences, $m_{ij(n)}$, produced from AND gates. Two kinds of synaptic weights are necessary for most feedforward networks: positive or excitatory weights, w_{ij}^{+}, and negative or inhibitory weights, w_{ij}^{-}. Two separate OR gates per neuron are, therefore, necessary to form excitatory and inhibitory net inputs.

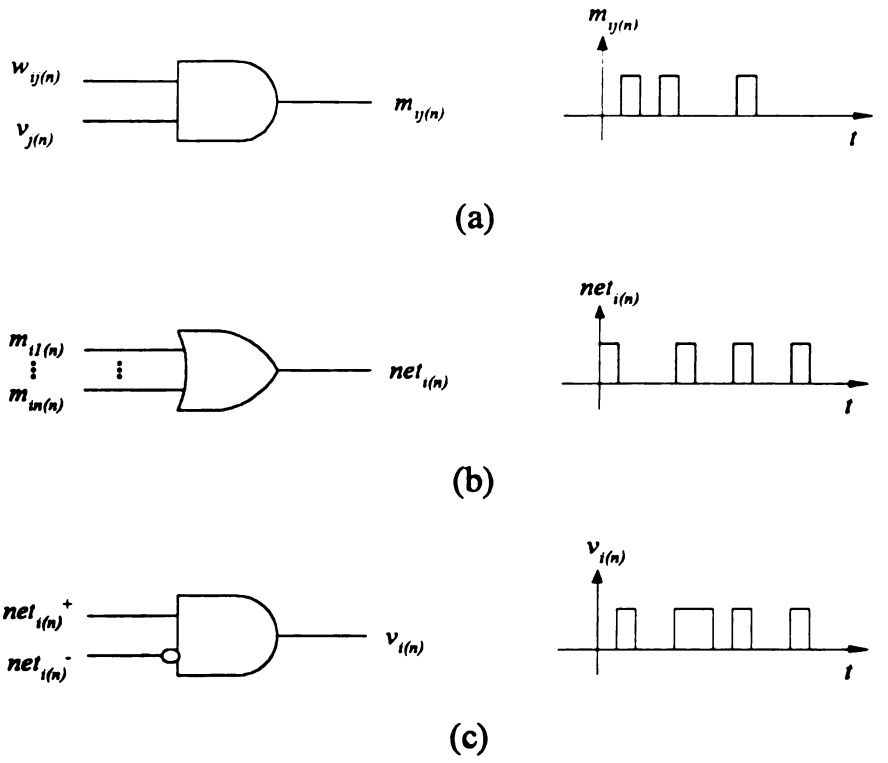


Figure 3.3. Stochastic Computation in the DMNN: (a) Stochastic Multiplication; (b) Logical OR; (c) Neural Activation [1].

Let net_i^+ be the probability of a pulse occurrence in the output sequence $net_{i(n)}^+$ of an n-input OR gate for excitatory net input in neuron i, and, net_i^- likewise for an inhibitory net input (Figure 3.3). net_i^+ and net_i^- can be described as

$$net_i^+ = P(net_{i(n)}^+ = 1) \tag{3.16}$$

$$= P\left(m_{i1(n)}^+ = 1 \vee m_{i2(n)}^+ = 1 \vee \ldots \vee m_{in(n)}^+ = 1\right) \tag{3.17}$$

$$= 1-\left(1-P\left(m_{i1(n)}^+ = 1\right)\right)\left(1-P\left(m_{i2(n)}^+ = 1\right)\right)\ldots\left(1-P\left(m_{in(n)}^+ = 1\right)\right) \tag{3.18}$$

$$= 1-\prod_{j=1}^{n}\left(1-m_{ij}^+\right) \tag{3.19}$$

$$= 1 - \prod_{j=1}^{n} \left(1 - w_{ij}^{-} v_j\right) \tag{3.20}$$

and $$net_i^{-} = 1 - \prod_{j=1}^{n} \left(1 + w_{ij}^{-} v_j\right) . \tag{3.21}$$

Two net inputs, formed by dedicated OR gates are ANDed together to form the activation function. The probability of a pulse occurrence, v_i, for two statistically independent sequences net_i^{+} and net_i^{-}, is given by

$$= P\left(net_{i(n)}^{+} = 1 \wedge net_{i(n)}^{-} = 0\right) \tag{3.22}$$

$$= net_i^{+}\left(1 - net_i^{-}\right) \tag{3.23}$$

$$= \left[1 - \prod_{j=1}^{n} \left(1 - w_{ij}^{+} v_j\right)\right] \prod_{j=1}^{n} \left(1 + w_{ij}^{-} v_j\right) . \tag{3.24}$$

The nonlinear activation function in equation (3.24) is both continuous and differentiable and therefore a good candidate for backpropagation training [3]. This form of stochastic computation has been used by Kim [1] in the development of the DMNN architecture, a subject for the next section.

A detailed mathematical derivation of the backpropagation algorithm using the above stochastic function is covered in Chapter 4.

3.3 Hardware Implementation

3.3.1 Basic Computing Elements

Kim developed a modular network architecture by developing the basic building blocks in the form of a Random Pulse Generator (RPG), a Synaptic element (SYN), an Input Neuron Body element (INB), and a Regular Neuron Body element (RNB) [1].

3.3.1.1 Random Pulse Generator (RPG)

The RPG is comprised of a tapped LFSR and a digital comparator. The tapped LFSR is constructed using D flip-flops and XOR logic gates [1]. Figure 3.4(a) shows an 8th order LFSR, with a feedback function $f(x) = x_2 \oplus x_3 \oplus x_4 \oplus x_8$, as implemented by XOR logic gates. The period of the sequence $v_{(n)}$ is $2^8-1 = 255$. Figure 3.4(b) presents an RPG. A logic '1' or '0' pulse is generated at every clock pulse for $v_i \geq x$ and $v_i < x$, respectively.

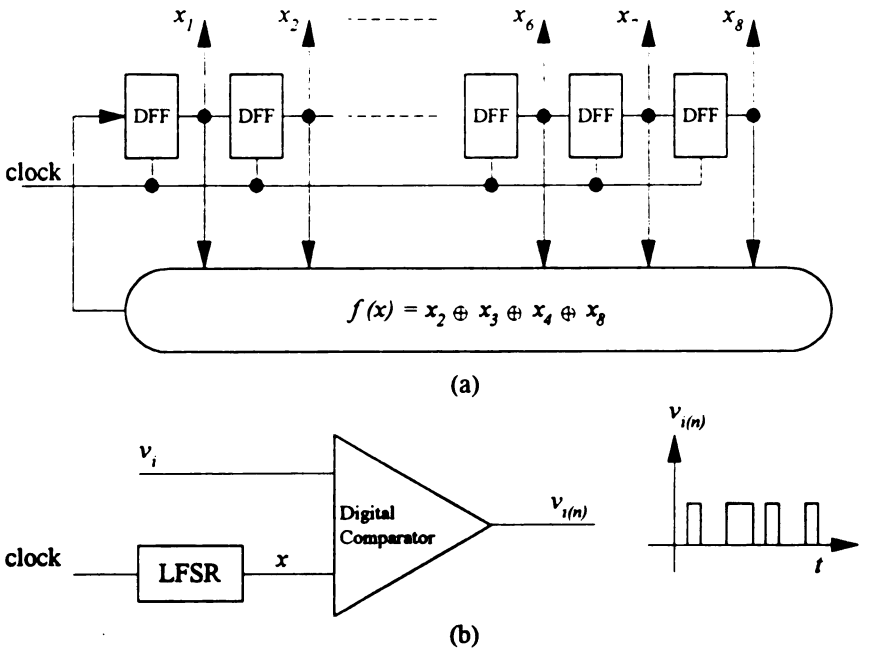


Figure 3.4. (a) An 8-bit LFSR with a Primitive Polynomial; (b) An RPG for v_i [1].

3.3.1.2 Synaptic Element

The synaptic element (SYN) consists of an RPG, a weight register, two AND gates, and two wired-OR lines [1]. Figure 3.5 shows the structure and block diagram of a digital synaptic element. Weight w_{ij} represented as an r-bit fractional number is loaded into a weight register and the corresponding random pulse stream $w_{ij(n)}$ is generated through the RPG. The pulse stream is transmitted to the two AND gates for discrimination of positive and negative weights. If the synaptic weight is positive, a resulting product sequence $m_{ij(n)}^{+}$ is transmitted to an excitatory net-input line. Otherwise, $m_{ij(n)}^{-}$ is transmitted to an inhibitory net-input line.

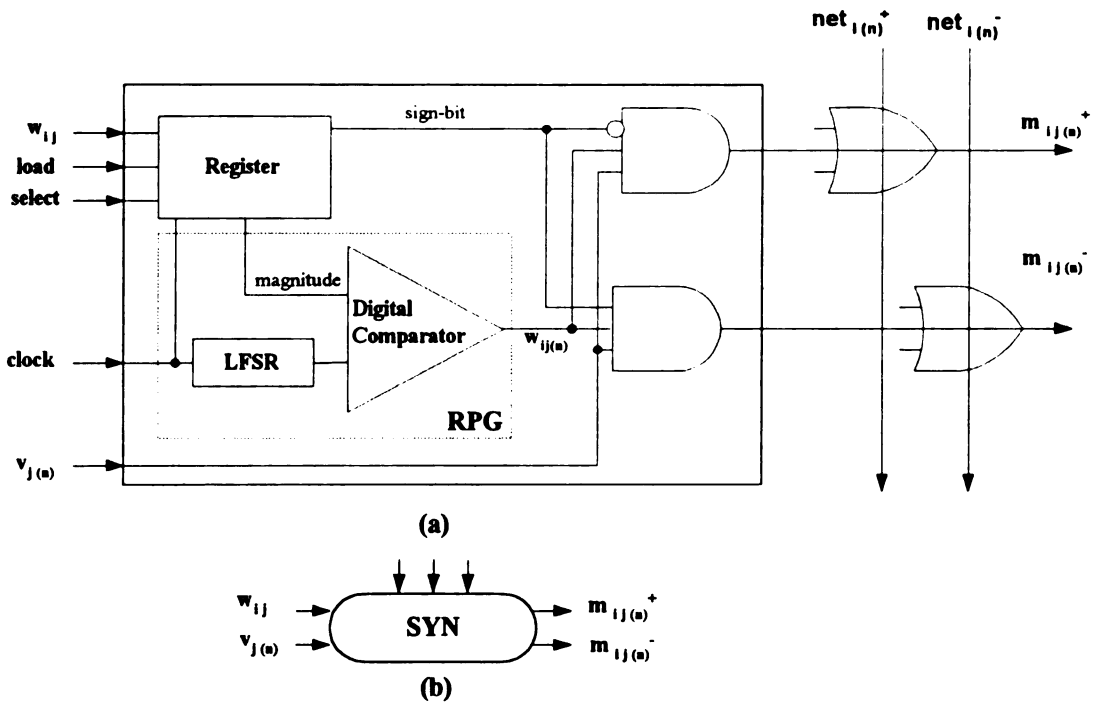


Figure 3.5. (a) A Synaptic Element; (b) Block Diagram of SYN [1].

3.3.1.3 Input Neuron Body Element (INB)

An INB consists of an n-bit register and an RPG [1]. The INB converts the value of the *i*th element in an input pattern, v_i, to a corresponding random pulse sequence $v_{i(n)}$. Figure 3.6 shows the structure and block diagram of the INB.

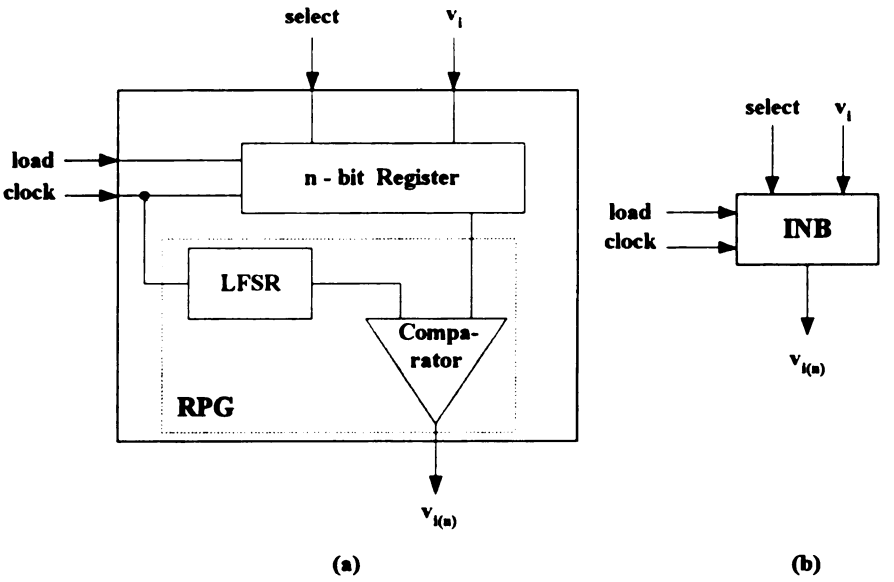


Figure 3.6. (a) Input Neuron Body (INB) Element; (b) Block Diagram of INB [1].

3.3.1.4 Regular Neuron Body Element (RNB)

An RNB consists of an AND gate, an OR gate, an up-counter, a 2x1 multiplexer, a buffer, and an RPG [1]. Two net-input pulse streams, transmitted from synaptic elements, are collected in an up-counter through an AND gate to form a neural activation. Figure 3.7 shows the structure and block diagram of an RNB.

v_i is estimated as $\hat{v}_i$ which is actually the value of the up-counter after each iteration. The signal *new_iter* changes from '0' to '1' after each iteration and transfers the output of the counter to a buffer via a 2x1 multiplexer at the next clock. At the same time, the up-counter is reset. This output is used to generate a new action pulse sequence $v_{i(n)}$ while the up-counter continues to accumulate incoming pulses. $\hat{v}_i$ (dotted line) is used

as an output of a neuron i in the output layer, while $v_{i(n)}$ (solid line) is used in the hidden layers.

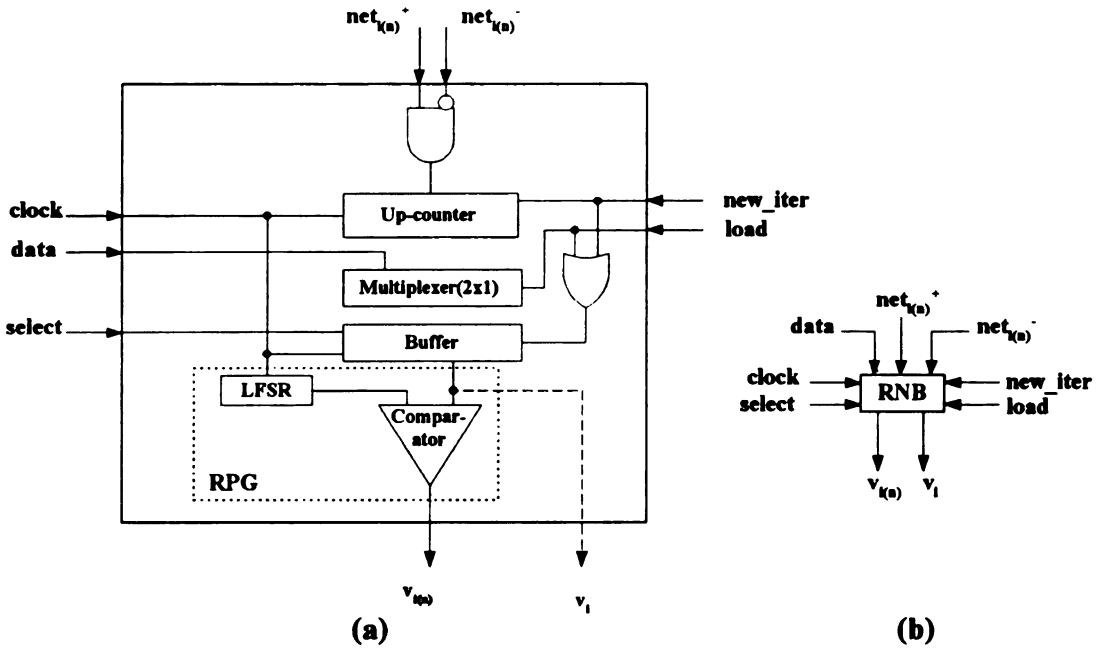


Figure 3.7. (a) A Regular Neuron Body (RNB) Element; (b) Block Diagram of RNB [1].

3.3.2 DMNN Architecture

The DMNN has been constructed using four basic modules: Input Layer Module (ILM), Synaptic Array Module (SAM), Regular Neuron-body Array Module (RNAM), and Interconnection Module (ICM) [1].

The ILM is composed of a group of INB elements. It receives input and transforms them into corresponding binary pulse sequences. These pulse sequences are transmitted to the SAM through the ICM. The SAM consists of a group of synaptic

elements and net input lines. All synaptic multiplications in the same layer are performed simultaneously. The ICM is a group of connection lines that transmit action pulse sequences from the previous layer to the SAM in the next layer. Pulses on two net-input sequences transmitted from the SAM are collected at the RNAM. All $v_{i(n)}$'s from the RNAM are produced simultaneously. Using these modules, any size of DMNN, with any number of neurons in each layer and any number of layers, can be configured. Figure 3.8 portrays architecture of the DMNN.

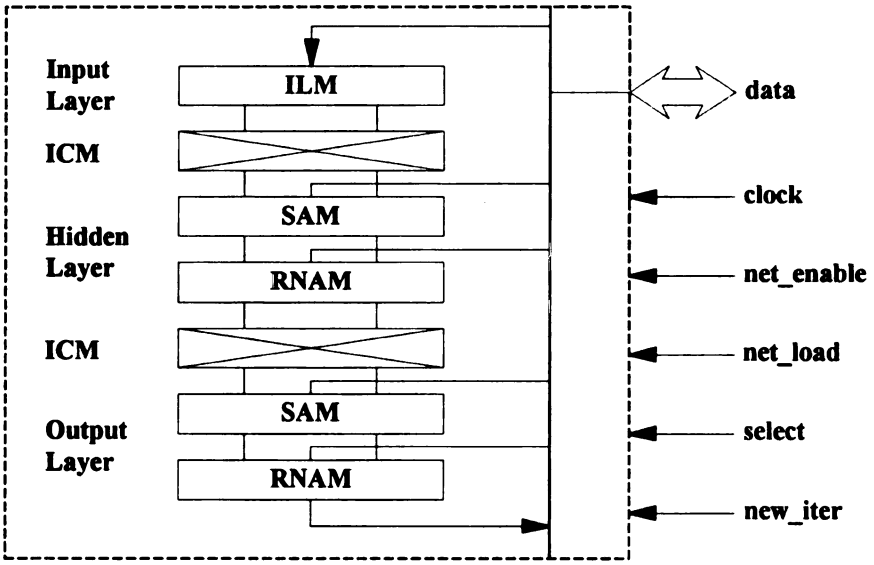


Figure 3.8. DMNN Architecture [1].

3.3.3 DMNN Coprocessor

The DMNN coprocessor is composed of a controller, a memory, an iteration counter and a clock generator. Figure 3.9 shows the architecture of a DMNN

coprocessor. The controller consists of a microprocessor and a control unit which may be either programmed or hardwired. The training of the network is performed on a host computer. The trained weights, network configuration, input patterns and some control commands are downloaded from the host memory.

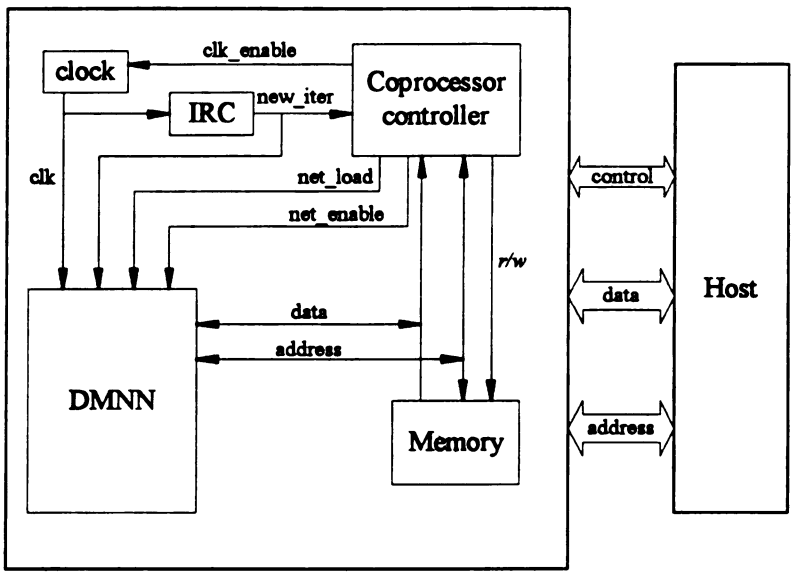


Figure 3.9. DMNN Coprocessor [1].

3.4 Behavioral Model of a DMNN Coprocessor

3.4.1 Design Methodology

The design of any hardware typically starts from a gate or circuit level schematic. However, as the system becomes complex, a top down approach is needed to reduce the development cost as well as the design cost. The top down design starts with a high level

specification which is subsequently reduced to lower level specifications in a hierarchical manner. Hardware Description Languages (HDL) are very important tools for a high level or top down design [46]. VHSIC Hardware Description Language (VHDL) is a typical behavioral description language which is semantically oriented for digital systems. VHDL can be used to model the behavior of systems and simulate them to verify the design. Modeling involves specifying the inputs and outputs of a device, and describing its behavior and/or structure [47].

A top down approach was needed to model the DMNN coprocessor. Kim used VHDL to model the proposed DMNN architecture. The coprocessor model starts with a high level specification of the network which is decomposed into lower level specifications in a hierarchical fashion [1]. Figure 3.10 shows the design hierarchy of the DMNN coprocessor. The DMNN itself consists of three layers: input layer, hidden layer, and an output layer. Each component is described in VHDL using the two styles of description: behavioral and structural. The components in the lower branches of the hierarchy are modeled using behavioral descriptions and all the other components are modeled using structural descriptions.

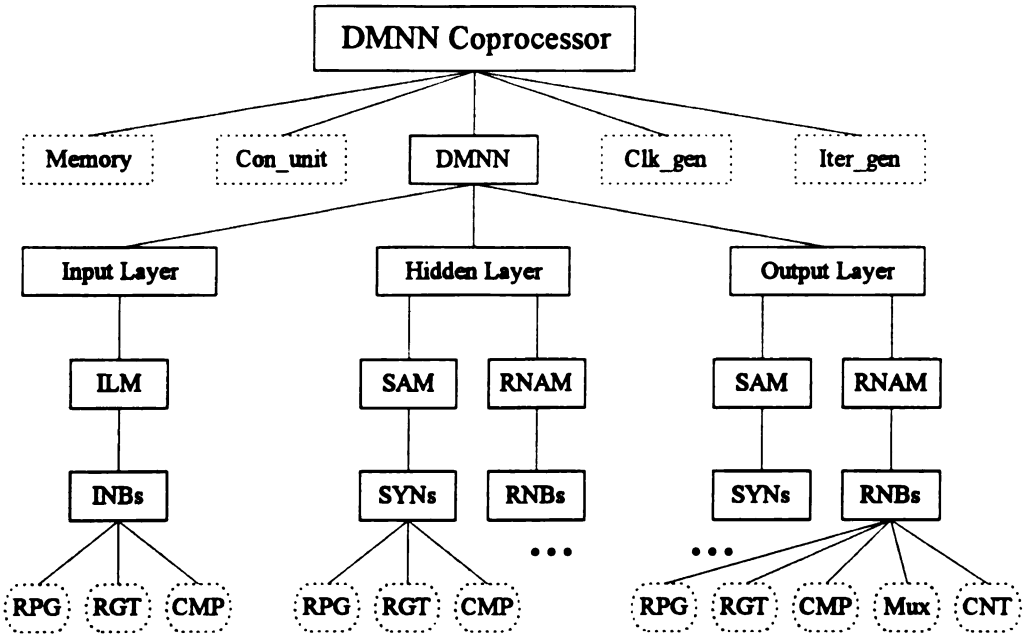


Figure 3.10. DMNN Structural Hierarchy [1].

3.4.2 Coprocessor Control

Kim's approach for modeling the DMNN coprocessor is elaborated below:

Step 1. The network is trained using software simulations and the backpropagation rule. (A detailed derivation of the backpropagation algorithm, using Kim's stochastic model, is presented in Chapter 4). The trained weights, input patterns and the test patterns are written in files.

Step 2. The DMNN coprocessor reads the weights file and the file for test patterns.

Step 3. The coprocessor initializes the LFSRs in the network and loads synaptic weight registers with the trained weight values.

Step 4. The coprocessor initiates the controller to start generating the control signals.

Step 5. The DMNN starts classifying the input patterns and writes the output patterns into a file.

The complete VHDL code for the DMNN coprocessor is given in [1].

3.5 DMNN Applications

3.5.1 Application Methodology

Binary classification problems are applied to the DMNN coprocessor as test bench problems. The sampling data, comprised of the training data and the testing data, is selected first for each problem. The DMNN is trained off-line on a host computer using the training data. The training is carried out using the backpropagation algorithm (Chapter 4). After the training is complete, the network architecture is setup and the test data is applied to the DMNN coprocessor. The performance of the DMNN is evaluated based on the test results.

Each pattern is represented as an array of binary numbers. The number of neurons in the input and output layers of the DMNN are directly dependent on the nature of the problem. The number of neurons in the input layer are one more than the number of elements in the pattern vector. The additional input is used to provide a permanent 1 as a threshold input to the synaptic elements located in the upper layers. The number of neurons in the output layer is the same as the number of categories into which the sampling patterns are to be classified.

Kim's results show that two layers (one hidden layer) are enough for a binary classification. He has also shown that a relationship exists between the existence of solutions, the value of the sum squared error and the ratio of input vs output values. Kim has suggested the ratio value as 0.1 and used 0.55 and 0.45 as target values to represent the ON and OFF states of the output neurons. The network configuration is a matrix that represents the interconnection of all the neuron elements in the network. It can also be represented as n_0 x n_1 x n_2 . . . x n_p, where n_0 denote the number of elements in the input layer, $n_1, n_2, \ldots, n_{p-1}$, denote the number of elements in the hidden layers and n_p represents the number of elements in the output layer. A minimal network configuration is defined as the network that converges to an error state equal to or below the predefined error threshold with a minimum number of neurons in the hidden layers and minimum number of hidden layers.

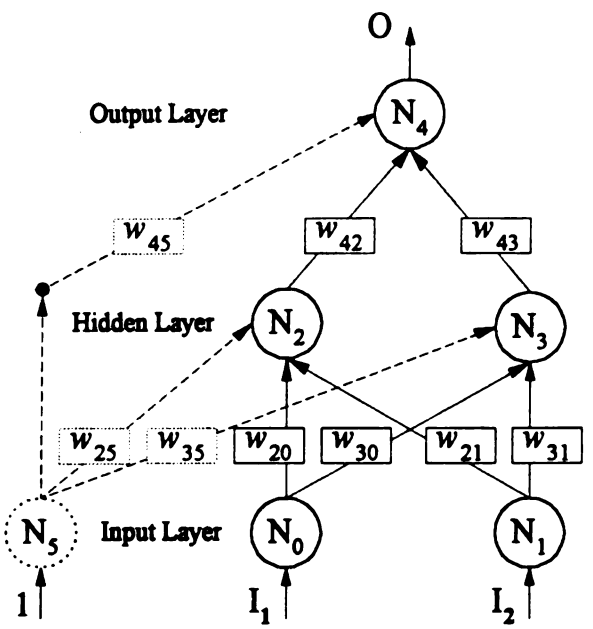
3.5.2 Benchmark Problems

Two benchmark problems: the XOR problem and an 8x3 encoding problem were applied to the DMNN coprocessor to test its ability to classify linearly unseparable patterns and whether it can work as a data compressor.

3.5.2.1 XOR Problem Solver

The Exclusive-OR (XOR) problem is a dual input single output problem with linearly unseparable patterns. Figure 3.11 displays an XOR problem. The nonlinearity associated with the XOR is also represented graphically. A network configuration of

3x2x1 was considered. Table 3.1 represents XOR problem for a DMNN showing four input patterns and their corresponding target patterns. The DMNN was able to classify the patterns correctly when the register length was more than '6'. Values below '6' degraded the network performance considerably.



Truth Table - XOR

I_1	I_2	O
0.0	0.0	0.45
0.0	1.0	0.55
1.0	0.0	0.55
1.0	1.0	0.45

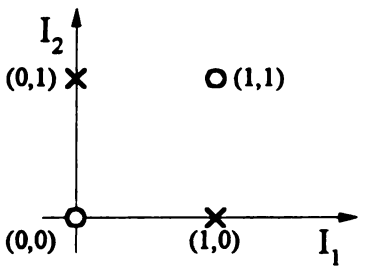


Figure 3.11. The XOR Problem.

Table 3.1. An n-bit Two-layer DMNN XOR Results [1].

		O					
I_1	I_2	n=10	n=9	n=8	n=7	n=6	n=5
0.0	0.0	0.451	0.456	0.461	0.468	0.469	0.500
0.0	1.0	0.549	0.556	0.563	0.563	0.688	0.563
1.0	0.0	0.549	0.556	0.570	0.578	0.719	0.563
1.0	1.0	0.457	0.449	0.445	0.484	0.625	0.438

3.5.2.2 8x3 Encoder

In order to see the behavior of DMNN as a data compressor, an 8x3 encoding problem was applied. A minimal network configuration of 9x3x3 was considered. Figure 3.12 portrays an example of the 8x3 encoder. The problem is represented in Table 3.2. The DMNN successfully encodes the 8-bit data when the register length is more than '8'.

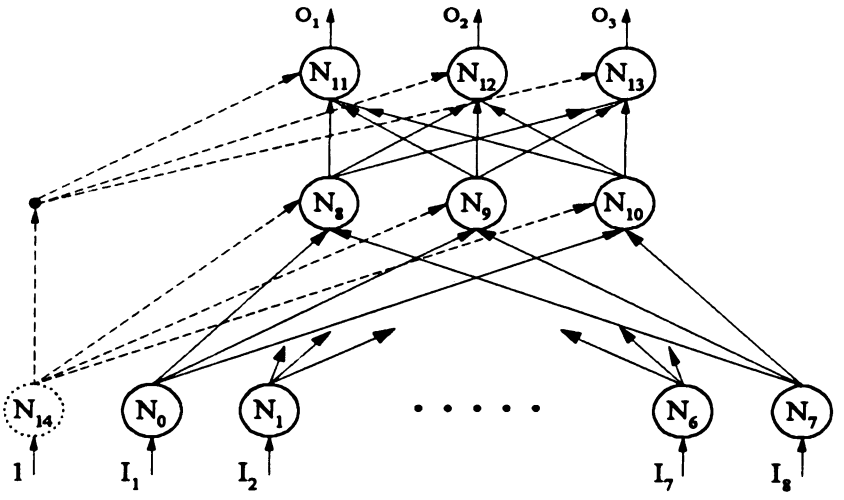


Figure 3.12. 8x3 Encoder.

Table 3.2. Truth Table for an 8x3 Encoder [1].

I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	O_1	O_2	O_3
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.45	0.45	0.45
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.55	0.45	0.45
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.45	0.55	0.45
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.55	0.55	0.45
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.45	0.45	0.55
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.55	0.45	0.55
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.45	0.55	0.55
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.55	0.55	0.55

3.5.3 DMNN Character Recognizer

The DMNN was used as a character recognizer for solving two problems: a five-digit problem for classifying digits 0 to 4 and a ten-digit classifier for digits 0 to 9. The network was trained with two sets of training data for each problem. The first set consists of ideal digits and the second set contained 10% noisy patterns (three patterns per digit). An additional set of 20% noisy patterns (three patterns per digit) was added in the test data set. For the five-digit classifier, each of the five digits were represented by a matrix of 6x4 pixels whereas the pixel size was increased to 7x5 in the ten-digit problem. Figures 3.13 and 3.14 depict the pixel images for the five-digit and ten-digit problems.

Kim compared the performance of the DMNN with an ordinary multilayer neural network employing a sigmoidal function. Typical results of his comparison using a 5-digit classifier are shown in Table 3.3. The '*' values indicate minimal network configuration and N1 and N2 are the number of neurons in the first and second hidden layers respectively. Each figure in the table is the average value based on ten different initial sets of weights. A learning rate (η) = 0.2, momentum factor (α) = 0.5, and sum squared error (ss-error) = 0.0001 are used for the DMNN while η = 1.0, α = 0.5, and ss-error = 0.01 are used for the ordinary neural network. Lower values of η and ss-error used for training of the DMNN are due to the constraints in its operation range.

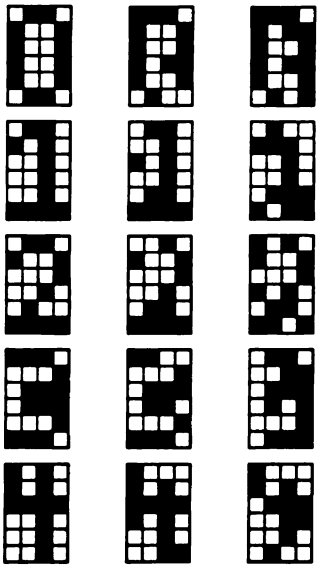


Figure 3.13. 5-Digit Classifier Patterns: (a) Ideal; (b) 10% Noisy; (c) 20% Noisy [1].

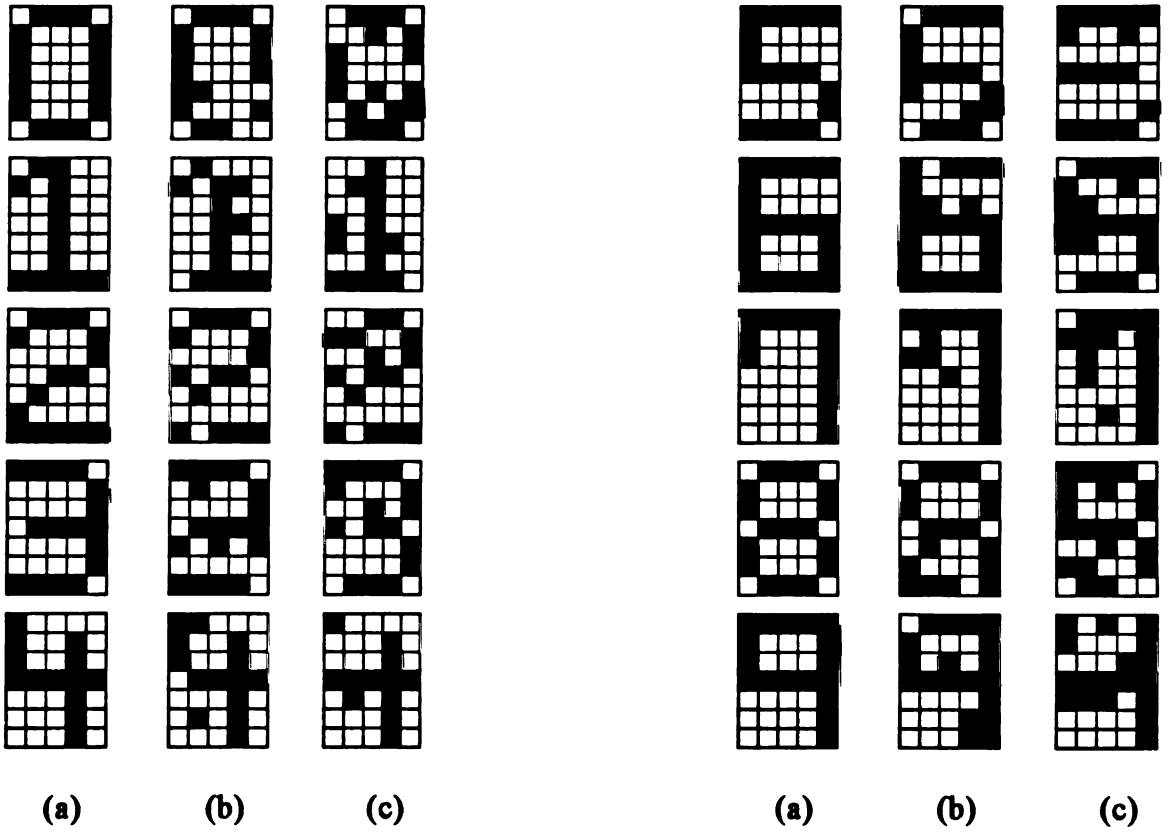


Figure 3.14. 10-Digit Classifier Patterns: (a) Ideal; (b) 10%Noisy; (c) 20% Noisy [1].

Table 3.3. Average Number of Iterations for 5-Digit Classifier [1].

DMNN						ANN		
Trained with $\alpha = 0.0$			Trained with $\alpha = 0.5$			Trained with $\alpha = 0.5$		
N_1	N_2	*No of iterations*	N_1	N_2	*No of iterations*	N_1	N_2	*No of iterations*
*4	0	2086	*4	0	1738	*5	0	13311
10	0	420	10	0	293	10	0	17212
30	0	332	30	0	182	15	0	11905
50	0	274	50	0	135	25	0	14108
6	4	12263	6	4	7482	8	4	79320
8	6	10446	8	6	4959	8	8	35480
10	8	7427	10	8	2972	10	8	44520
12	10	6135	12	10	1820	12	10	55025

Kim's results show that the DMNN network can be trained with a significantly reduced number of iterations compared to an ordinary ANN. The fast convergence obtained in the DMNN is due to the fact that the values of the output neurons can take on intermediate values between 0 and 1 as target values. The number of iterations generally increase with the number of hidden layers and decrease with the increase in the number of neurons in the hidden layers of the DMNN.

Performance results for a 5-digit recognizer are tabulated in Table 3.4. A DMNN model with 8-bit and 9-bit register lengths and a C simulated DMNN are compared for misclassification errors. The misclassification occurs when no neuron or more than one neurons are turned on during the test mode. Each figure is based on an average result of 20 tests and for each test a new set of synaptic weights are loaded. The results indicate that a 9-bit character recognizer behaves very close to a deterministic simulation. It is also noted that the performance of DMNN improves when training data contains noisy patterns.

Table 3.4. Performance of 5-Digit Classifier [1].

	Trained with ideal digits		Trained with ideal + 10% noisy data
	Misclassified error on 10% noisy data	*Misclassified error on 20% noisy data*	*Misclassified error on 20% noisy data*
8-Bit DMNN	32.2%	55.6%	38.7%
9-Bit DMNN	11.7%	30%	26.7%
C-simulation	8.5%	43%	35.3%

Table 3.5 shows the performance results for a 10-digit classifier. The learning rate (η) = 0.05, momentum factor (α) = 0.0, and ss-error = 0.005 are used. Only two layers are considered for this experiment and minimal network configurations are 36x9x10 and 36x30x10 for ideal set and ideal plus noisy sets of training data, respectively. The results are compared with those of the deterministic simulations and ordinary ANN classifier. Each figure in the table is the average value based on twenty sets of weights. The classification rates obtained from the DMNN are close to the results of the deterministic simulations when the register length is greater than 8.

Table 3.5. Performance of 10-Digit Classifier [1].

	Trained with ideal digits		Trained with ideal + 10% noisy data
	Misclassification error on 10% noisy data	*Misclassification error on 20% noisy data*	*Misclassification error on 20% noisy data*
8-bit DMNN	16.2%	43.3%	25.4%
9-bit DMNN	9.7%	36.6%	20.3%
C-simulation	6.7%	35.0%	13.5%
ANN	15.8%	30.8%	16.7%

3.6 Performance Summary

A pulse mode digital multilayer neural network architecture implementable with simple logic gates, developed by Kim, has been presented. The DMNN employs stochastic computing techniques and a modular architecture that is compact and programmable. All operations in a layer are performed in parallel and operations between two layers are performed in a pipeline fashion thus utilizing full parallelism. The processing speed depends only on the clock speed and the register length; not on the network size. Therefore, any size of network can be constructed by interconnecting the desired number of modules without compromising speed. The DMNN has been applied successfully to binary classification problems such as XOR, 8x3 encoding, and character recognition. Kim simulated the DMNN in VHDL and found the performance comparable to deterministic DMNN simulations and ordinary backpropagation neural networks.

CHAPTER 4

BACKPROPAGATION IN THE DMNN

4.1 Introduction

The digital multilayer neural network is a feedforward neural network which can be trained with the backpropagation algorithm discussed in Section 2.3.4.2.2. Backpropagation performs an iterative gradient descent over a sum-squared error measure. This chapter discusses the method of implementing the non traditional stochastic function, used by Kim (Chapter 3), into the backpropagation algorithm. Kim, though he uses the backpropagation algorithm in his DMNN, does not provide the mathematical details of the C simulation program [1]. In order to understand his C code and to be able to develop an improved simulation technique, it is essential to derive the backpropagation equations. These equations are subsequently used in the software simulation of the DMNN.

The backpropagation technique is criticized and its inherent weaknesses are identified. Enhancements are suggested in the end to improve the performance of the DMNN.

4.2

4.2.1

This m

develo

simula

2.3.4.

preser

4.2 Backpropagation Implementation

4.2.1 Review of the Stochastic Function

The stochastic function developed by Kim [1] has been presented in Section 3.1.3. This non-linear, non-decreasing, continuously differentiable function has been used in the development of a digital multilayer neural network. This section discusses a software simulation of the DMNN using the backpropagation algorithm discussed in Section 2.3.4.2.2. To review the stochastic function, a three layered feedforward neural network presented in Figure 2.8, is redrawn again as Figure 4.1 for ease of reference.

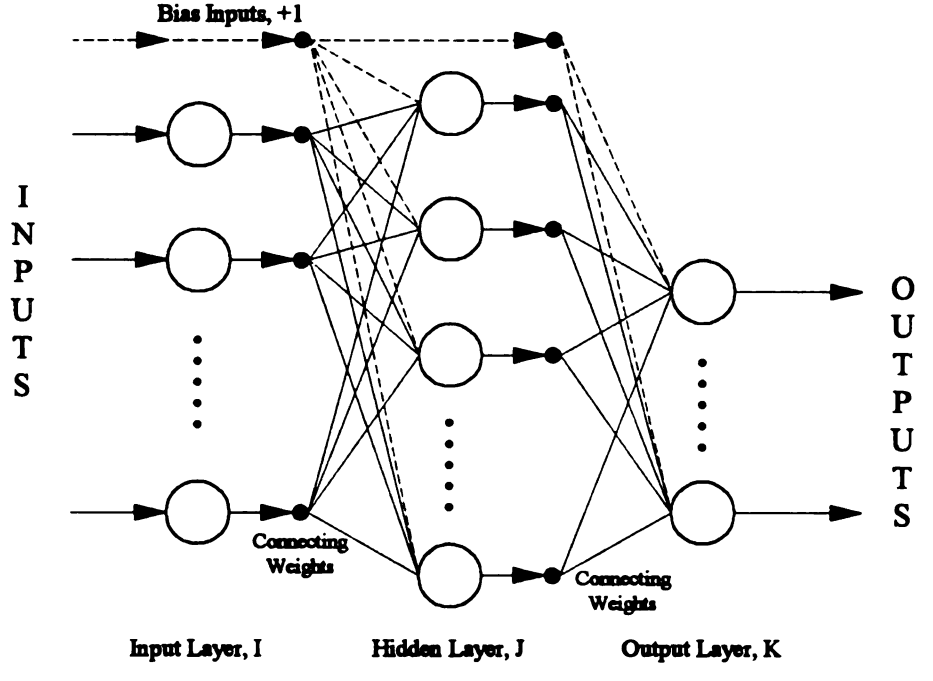


Figure 4.1. Three Layered Neural Network.

In the above figure, v_i is the output of the ith input layer passive neuron, v_j is the output of the jth hidden layer active neuron, and v_k is the output of the of the kth output layer neuron. w_{ij}'s are the synaptic weights connecting the ith input layer neuron to the jth hidden layer neurons and w_{jk}'s are the synaptic weights connecting jth hidden layer neuron to the kth output layer neurons.

From Section 3.2.3, we know that the excitatory net input for the jth neuron in the hidden layer is given by

$$net_j^+ = 1 - \prod_{i=0}^{ni} \left(1 - w_{ij}^+ v_i\right) \tag{4.1}$$

and the inhibitory net input for the jth neuron in the hidden layer is given by

$$net_j^- = 1 - \prod_{i=0}^{ni} \left(1 + w_{ij}^- v_i\right). \tag{4.2}$$

In equations (4.1) and (4.2), $v_i = 1$ for $i = 0$, and n_i is the total number of neurons in the input layer.

Similarly, the excitatory net input for kth neuron in the output layer is

$$net_k^+ = 1 - \prod_{j=0}^{nj} \left(1 - w_{jk}^+ v_j\right) \tag{4.3}$$

and the inhibitory net input for the kth neuron in the output layer is

$$net_k^- = 1 - \prod_{j=0}^{nj} \left(1 + w_{jk}^- v_j\right). \tag{4.4}$$

In equations (4.3) and (4.4), $v_j = 1$ for $j = 0$, and n_j is the total number of neurons in the hidden layer. Therefore the output of the kth neuron in the output layer is

$$v_k = net_k^+\left(1 - net_k^-\right) . \tag{4.5}$$

From equations (4.3) and (4.4)

$$v_k = \left[1 - \prod_{j=0}^{n_j}\left(1 - w_{jk}^+ v_j\right)\right] \prod_{j=0}^{n_j}\left(1 + w_{jk}^- v_j\right) \tag{4.6}$$

where v_j is the output of the jth hidden layer neuron given by

$$v_j = net_j^+\left(1 - net_j^-\right) . \tag{4.7}$$

Finally, from equations (4.1) and (4.2)

$$v_j = \left[1 - \prod_{i=0}^{n_i}\left(1 - w_{ij}^+ v_i\right)\right] \prod_{i=0}^{n_i}\left(1 + w_{ij}^- v_i\right) . \tag{4.8}$$

4.2.2 Backpropagation Analysis

The outputs, v_k for the output layer and v_j for the hidden layer, are calculated using equations (4.1) and (4.3) for positive weights and equations (4.2) and (4.4) for negative weights. Let t_k be the desired outputs. The objective is to minimize the difference between the desired outputs, t_k, and the actual outputs v_k.

Define the error cost function E_k for each k as

$$E_k = \frac{1}{2}\left(t_k - v_k\right)^2 . \tag{4.9}$$

Therefore, minimizing the cost function for the kth output node with respect to the weight connecting jth hidden node involves using

$$\frac{\partial E_k}{\partial w_{jk}^+} = \frac{\partial E_k}{\partial v_k} \bullet \frac{\partial v_k}{\partial w_{jk}^+} \tag{4.10}$$

and

$$\frac{\partial E_k}{\partial w_{jk}^-} = \frac{\partial E_k}{\partial v_k} \bullet \frac{\partial v_k}{\partial w_{jk}^-} . \tag{4.11}$$

From equations (4.9) and (4.10)

$$\frac{\partial E_k}{\partial w_{jk}^+} = -\left(t_k - v_k\right)\frac{\partial v_k}{\partial w_{jk}^+} \tag{4.12}$$

and from equations (4.9) and (4.11)

$$\frac{\partial E_k}{\partial w_{jk}^-} = -\left(t_k - v_k\right)\frac{\partial v_k}{\partial w_{jk}^-} . \tag{4.13}$$

Applying the chain rule gives

$$\frac{\partial v_k}{\partial w_{jk}^+} = \frac{\partial v_k}{\partial net_k^+} \bullet \frac{\partial net_k^+}{\partial w_{jk}^+} \tag{4.14}$$

and

$$\frac{\partial v_k}{\partial w_{jk}^-} = \frac{\partial v_k}{\partial net_k^-} \bullet \frac{\partial net_k^-}{\partial w_{jk}^-} . \tag{4.15}$$

From equation (4.5), we find

$$\frac{\partial v_k}{\partial net_k^+} = 1 - net_k^- \tag{4.16a}$$

and

$$\frac{\partial v_k}{\partial net_k^-} = -net_k^+ . \tag{4.16b}$$

From equations (4.3) and (4.4), we obtain

$$\frac{\partial net_k^+}{\partial w_{jk}^+} = \left(1 - net_k^+\right)\frac{v_j}{1 - w_{jk}^+ v_j} \tag{4.17}$$

where, $v_j = 1$ for $j = 0$, and

$$\frac{\partial net_k^-}{\partial w_{jk}^-} = -\left(1 - net_k^-\right)\frac{v_j}{1 + w_{jk}^- v_j} \tag{4.18}$$

where, $v_j = 1$ for $j = 0$.

Now backtracking by substituting equations (4.16), (4.17), (4.18) in equations (4.14) and (4.15) and then substituting equations (4.14) and (4.15) in equations (4.12) and (4.13) at appropriate places, we obtain

$$\frac{\partial E_k}{\partial w_{jk}^+} = -\left(t_k - v_k\right)\left(1 - net_k^-\right)\left(1 - net_k^+\right)\frac{v_j}{1 - w_{jk}^+ v_j} \tag{4.19}$$

and

$$\frac{\partial E_k}{\partial w_{jk}^-} = -\left(t_k - v_k\right)\left(net_k^+\right)\left(1 - net_k^-\right)\frac{v_j}{1 + w_{jk}^- v_j} . \tag{4.20}$$

Let

$$\delta_k^+ = \left(t_k - v_k\right)\left(1 - net_k^-\right) \tag{4.21}$$

and

$$\delta_k^- = \left(t_k - v_k\right)\left(net_k^+\right) . \tag{4.22}$$

Hence equation (4.19) becomes

$$\frac{\partial E_k}{\partial w_{jk}^+} = -\delta_k^+\left(1-net_k^+\right)\frac{v_j}{1-w_{jk}^+ v_j} \tag{4.23}$$

and from equation (4.20)

$$\frac{\partial E_k}{\partial w_{jk}^-} = -\delta_k^-\left(1-net_k^-\right)\frac{v_j}{1+w_{jk}^- v_j} \tag{4.24}$$

where, $v_j = 1$, for $j = 0$.

Backpropagation is a gradient descent learning rule. For excitatory weights, it is given by

$$w_{jk}^+[n+1] = w_{jk}^+[n] + \eta\left(-\frac{\partial E_k}{\partial w_{jk}^+}\right) \tag{4.25}$$

and for inhibitory weights, by

$$w_{jk}^-[n+1] = w_{jk}^-[n] + \eta\left(-\frac{\partial E_k}{\partial w_{jk}^-}\right) \tag{4.26}$$

where n is the number of presentation and η is the learning rate. Substituting equations (4.23) and (4.24) in equations (4.25) and (4.26) respectively gives

$$w_{jk}^+[n+1] = w_{jk}^+[n] + \eta\delta_k^+\left(1-net_k^+\right)\frac{v_j}{1-w_{jk}^+ v_j} \tag{4.27}$$

and

$$w_{jk}^-[n+1] = w_{jk}^-[n] - \eta\delta_k^-\left(1-net_k^-\right)\frac{v_j}{1+w_{jk}^- v_j} \tag{4.28}$$

where, $v_j = 1$ for $j = 0$.

It is important to note that equations (4.27) and (4.28) are only valid for weights connecting the *j*th hidden layer neurons to the *k*th neuron in the ouput layer. Weights connecting any other two layers are updated by reflecting back the error or difference at the output neurons. Therefore, referring back to Figure 4.1, weights, w_{ij} are updated reflecting back the error from all the output nodes to the *j*th hidden node.

That is,
$$w_{ij}^{+}[n+1] = w_{ij}^{+}[n] + \eta\left(-\sum_{k=1}^{n_k}\frac{\partial E_k}{\partial w_{ij}^{+}}\right) \tag{4.29}$$

and
$$w_{ij}^{-}[n+1] = w_{ij}^{-}[n] + \eta\left(-\sum_{k=1}^{n_k}\frac{\partial E_k}{\partial w_{ij}^{-}}\right). \tag{4.30}$$

Using chain rule gives

$$\sum_{k=1}^{n_k}\frac{\partial E_k}{\partial w_{ij}^{+}} = \sum_{k=1}^{n_k}\frac{\partial E_k}{\partial v_k}\bullet\frac{\partial v_k}{\partial net_k^{+}}\bullet\frac{\partial net_k^{+}}{\partial v_j}\bullet\frac{\partial v_j}{\partial net_j^{+}}\bullet\frac{\partial net_j^{+}}{\partial w_{ij}^{+}} \tag{4.31}$$

and similarly
$$\sum_{k=1}^{n_k}\frac{\partial E_k}{\partial w_{ij}^{-}} = \sum_{k=1}^{n_k}\frac{\partial E_k}{\partial v_k}\bullet\frac{\partial v_k}{\partial net_k^{-}}\bullet\frac{\partial net_k^{-}}{\partial v_j}\bullet\frac{\partial v_j}{\partial net_j^{-}}\bullet\frac{\partial net_j^{-}}{\partial w_{ij}^{-}}. \tag{4.32}$$

We know that
$$\frac{\partial E_k}{\partial v_k} = -\left(t_k - v_k\right), \tag{4.33}$$

$$\frac{\partial v_k}{\partial net_k^{+}} = 1 - net_k^{-}, \tag{4.34a}$$

and
$$\frac{\partial v_k}{\partial net_k^{-}} = -net_k^{+}. \tag{4.34b}$$

Therefore, $$\frac{\partial E_k}{\partial v_k} \bullet \frac{\partial v_k}{\partial net_k^+} = -\left(t_k - v_k\right)\left(1 - net_k^-\right) = -\delta_k^+ \tag{4.35}$$

and $$\frac{\partial E_k}{\partial v_k} \bullet \frac{\partial v_k}{\partial net_k^-} = -\left(t_k - v_k\right)\left(-net_k^+\right) = \delta_k^- \ . \tag{4.36}$$

Next, from equations (4.3) and (4.4) we write

$$\frac{\partial net_k^+}{\partial v_j} = \left(1 - net_k^+\right)\frac{w_{jk}^+}{1 - w_{jk}^+ v_j} \tag{4.37}$$

and $$\frac{\partial net_k^-}{\partial v_j} = -\left(1 - net_k^-\right)\frac{w_{jk}^-}{1 + w_{jk}^- v_j} \ . \tag{4.38}$$

Furthermore, from equation (4.7)

$$\frac{\partial v_j}{\partial net_j^+} = \left(1 - net_j^-\right) \tag{4.39a}$$

and $$\frac{\partial v_j}{\partial net_j^-} = -net_j^+ \tag{4.39b}$$

and finally from equations (4.1) and (4.2) we find

$$\frac{\partial net_j^+}{\partial w_{ij}^+} = \left(1 - net_j^+\right)\frac{v_i}{1 - w_{ij}^+ v_i} \tag{4.40}$$

and $$\frac{\partial net_j^-}{\partial w_{ij}^-} = -\left(1 - net_j^-\right)\frac{v_i}{1 + w_{ij}^- v_i} \ . \tag{4.41}$$

Substituting equations (4.33) through (4.41) in equations (4.31) and (4.32) at appropriate places, we obtain

$$\sum_{k=1}^{n_k} \frac{\partial E_k}{\partial w_{ij}^+} = -\sum_{k=1}^{n_k} \left(\delta_k^+\right)\left(1-net_k^+\right)\frac{w_{jk}^+}{1-w_{jk}^+ v_j}\left(1-net_j^-\right)\left(1-net_j^-\right)\frac{v_i}{1-w_{ij}^+ v_i} \tag{4.42}$$

and

$$\sum_{k=1}^{n_k} \frac{\partial E_k}{\partial w_{ij}^-} = -\sum_{k=1}^{n_k} \left(\delta_k^-\right)\left(1-net_k^-\right)\frac{w_{jk}^-}{1+w_{jk}^- v_j}\left(net_j^+\right)\left(1-net_j^-\right)\frac{v_i}{1+w_{ij}^- v_i} . \tag{4.43}$$

Now, define the identities

$$\delta_j^+ = \left(1-net_j^-\right)\sum_{k=1}^{n_k} \delta_k^+\left(1-net_k^+\right)\frac{w_{jk}^+}{1-w_{jk}^+ v_j} \tag{4.44}$$

and

$$\delta_j^- = net_j^+ \sum_{k=1}^{n_k} \delta_k^-\left(1-net_k^-\right)\frac{w_{jk}^-}{1+w_{jk}^- v_j} . \tag{4.45}$$

Hence, equations (4.42) and (4.43) become

$$\sum_{k=1}^{n_k} \frac{\partial E_k}{\partial w_{ij}^+} = -\delta_j^+\left(1-net_j^+\right)\frac{v_i}{1-w_{ij}^+ v_i} \tag{4.46}$$

and

$$\sum_{k=1}^{n_k} \frac{\partial E_k}{\partial w_{ij}^-} = -\delta_j^-\left(1-net_j^-\right)\frac{v_i}{1+w_{ij}^- v_i} . \tag{4.47}$$

Substituting equations (4.46) and (4.47) in equations (4.29) and (4.30) respectively, we obtain

$$w_{ij}^+[n+1] = w_{ij}^+[n] + \eta\delta_j^+\left(1-net_j^+\right)\frac{v_i}{1-w_{ij}^+ v_i} \tag{4.48}$$

and

$$w_{ij}^-[n+1] = w_{ij}^-[n] + \eta\delta_j^-\left(1-net_j^-\right)\frac{v_i}{1+w_{ij}^- v_i} . \tag{4.49}$$

Finally, rewrite the above equations as

$$\Delta w_{ij}^+[n+1] = \eta\delta_j^+\left(1-net_j^+\right)\frac{v_i}{1-w_{ij}^+ v_i} \tag{4.50}$$

and $$\Delta w_{ij}^{-}[n+1] = \eta \delta_j^{-} \left(1 - net_j^{-}\right) \frac{v_i}{1 + w_{ij}^{-} v_i} \tag{4.51}$$

where n is the number of presentation steps and $\Delta w_{ij}[n+1]$ is the change in weight required for the next presentation.

Flowcharts representing the implementation of DMNN using this backpropagation algorithm are placed in Appendix A.

4.3 Reviewing Backpropagation Philosophy

We know that each input vector V_i in a backpropagation algorithm has a corresponding desired output vector T. In response to V_i, the system yields its own output vector V_k. The backpropagation philosophy considers performance error to be the vector difference between V_k and T, that is, error vector E is

$$E = T - V_k = ([t_1 - v_{k1}], [t_2 - v_{k2}], \ldots , [t_n - v_{kn}]) . \tag{4.52}$$

Minimizing the squared length of the error vector E amounts to minimizing the sum of the squares of each of the elements or

$$|| E ||^2 = [t_1 - v_{k1}]^2 + [t_2 - v_{k2}]^2 + \ldots + [t_n - v_{kn}]^2 . \tag{4.53}$$

The main idea is to minimize E by adjusting each weight in the ANN. The technique is similar to the Widrow-Hoff algorithm for linear systems [48], even though an ANN is typically nonlinear. The entire approach rests on the desire to modify each weight by an amount proportional to how it can decrease the output error. In other words

$$\frac{dw}{dt} = -\eta \frac{d||E||^2}{dw} \tag{4.54}$$

where d*w*/dt is the amount of adjustment each weight would receive for one training exposure. Each weight is updated as

$$w[n+1] = w[n] + \frac{dw}{dt} . \tag{4.55}$$

The critical difference between the backpropagation and the Widrow-Hoff model is that the former uses PEs with nonlinear output transfer functions and the latter uses a linear function. This poses a problem for the backpropagation algorithm. In a linear system, any arrangement of rows of PEs with linear data processing could always be reduced to just two rows. As a consequence, error minimization amounts to adjusting those weights which will maximize the descent down an *n*-dimensional parabolic basin, the least mean-squared error surface. This makes implementation of equation (4.54) straight forward in linear systems [48]. Figure 4.2 portrays the least mean squared error surfaces in a linear and a nonlinear ANN.

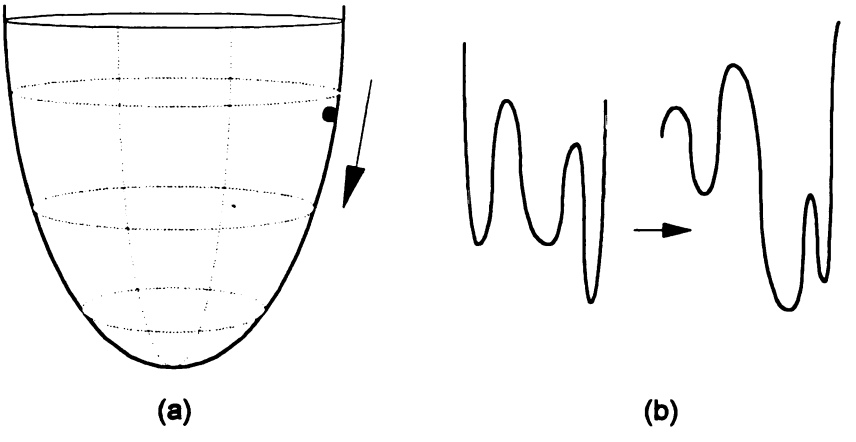


Figure 4.2. Error Surface in: (a) Linear ANN; (b) Nonlinear ANN.

An ANN with rows of nonlinear PEs cannot, in general, be reduced to fewer rows. This has two major consequences :

1. We must carry out the same algorithm for all rows when adjusting the weights.

2. The error surface is gutted with many basins of which few should ever resemble the analytically appealing paraboloid (Figure 4.2).

The consequences of these two issues as well as a few others [49] are examined in the following sections.

4.3.1 Backpropagation through Noise

Since, the system error is first derived at the output, the error signal must propagate back through the rows in order to reach every row. To begin with, however, every row is assigned random weights. Thus the error becomes increasingly meaningless as it backpropagates through the net of randomized weights. As a consequence, the front rows are 'misled' into what adjustment was truly needed. After this almost random correction at the front rows, the back rows will now receive data corrupted by the updated front rows. This noisy communication appears to be very inefficient because some simple training tasks may require thousands of training cycles.

4.3.2 Simultaneous Change

The mathematical derivation of the formula for weight adjustment uses partial derivatives, which assume that nothing else is being changed except the weight in question. However, in the true gradient descent method the weight change is affected after the complete set of training patterns have been presented to the network. If the weight change is to be implemented at the presentation of each pattern then the learning rate has to be small enough to enable minimization of the error function.

4.3.3 Numerous Basins and Crests

One reason why the mean-squared method works well for linear systems is that the error function yields only one *n*-dimensional parabolic error surface having only one minima at the paraboloid's bottom. Nonlinear systems, on the other hand, will typically have an error surface indented with numerous basins. For backpropagation, the problem is enhanced since these basins are usually non-parabolic and time varying as shown in Figure 4.3.

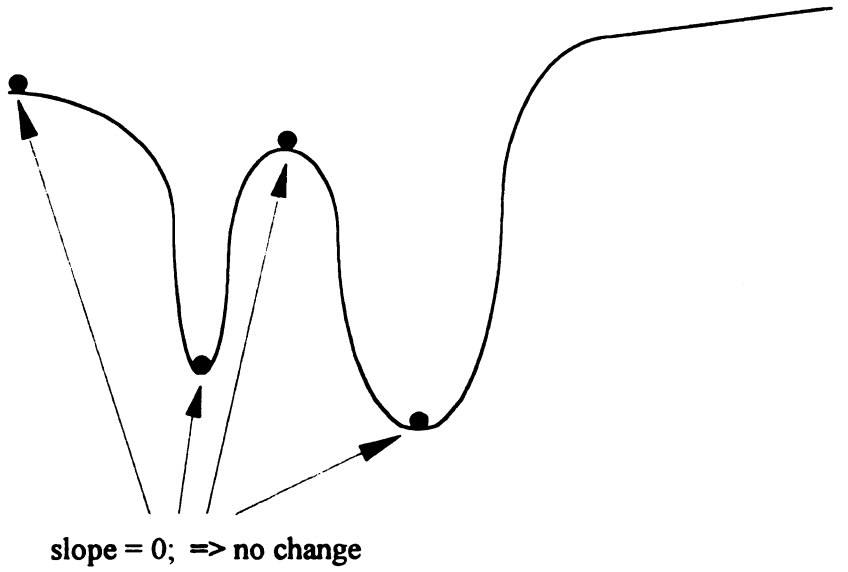


Figure 4.3. Local Minimas and Relative Weight Changes.

Zero slopes exist between any two basins as well as at the bottom of each basin (local minima). This implies that the weight modification may be very slow within regions of relative maximum error as well as at the relative minimas. This is yet another cause for slow learning. Since there is no simple relation between slope and distance to a relative minima, large misguided adjustments can frequently occur, sending the system state straight down through the local minima and far up the other side, giving the system an error performance worse than prior to the adjustment.

4.3.4 Lack of Digital Convergence Proof

It can be shown mathematically that if one uses infinitesimally small weight adjustments, which also requires infinitely long patience, the convergence to an optimal

behavior will occur in a backpropagation network. The reality, however, requires a finite training period and therefore a fixed valued learning constant. Thus, digital applications may not necessarily converge and there is no way to find out when to stop the training session.

4.3.5 Dependence on Initial Conditions

The presence of numerous minimas with unequal depths is another significant issue. Convergence towards one minima may yield a lower mean squared error than convergence towards another. There is no direct way to determine whether the backpropagation has reached a local minima or a global minima. The only way of reaching better results is to randomize the weights and re-train again and again. The final performance, therefore, is partially dependent upon the randomized initial conditions for training.

4.3.6 Non-real Time Adaptation

The backpropagation tends to fade its memory of previously learnt patterns as it learns the new patterns. The consequence is that every new pattern for training requires all the old patterns to be recycled alongwith inorder to achieve a long term noise immunity during the training

4.4 Accelerating the Convergence

Several heuristics are available for attaining improvements in the rate of convergence [3,49,50]. The methods adopted by Kim [1] are discussed, and variations are suggested to further accelerate the convergence process.

4.4.1 Momentum Factor (α)

Rumelhart suggests that one way to increase the learning rate without leading to oscillations is to modify the generalized delta rule to include a momentum term [3]. This can be accomplished by the rule

$$\Delta w_{ij}[n+1] = \eta\,(\delta_{pj} O_{pi}) + \alpha\,\Delta w_{ij}[n] \qquad (4.56)$$

where n is the presentation number, $\Delta w_{ij}[n+1]$ is the weight change required for the next step and α is the momentum factor such that $0 \le \alpha \le 1$. α determines the effect of past weight changes on the current direction of movement in the weight space. It provides each connection weight w_{ij} with a kind of momentum so that it tends to change in the direction of the average downhill force instead of oscillating with the high frequency variations of the error surface in the weight space. In turn, the effective learning rate η can be made larger without initiating oscillations.

Adding the momentum term in equations (4.50) and (4.51) gives

$$\Delta w_{ij}^{-}[n+1] = \eta \delta_j^{-}\left(1 - net_j^{-}\right)\frac{v_i}{1 - w_{ij}^{-} v_i} + \alpha\, \Delta w_{ij}^{-}[n] \tag{4.57}$$

and

$$\Delta w_{ij}^{-}[n+1] = \eta \delta_j^{-}\left(1 - net_j^{-}\right)\frac{v_i}{1 + w_{ij}^{-} v_i} + \alpha\, \Delta w_{ij}^{-}[n] \; . \tag{4.58}$$

4.4.1.1 Implementing the Momentum Factor

Kim uses following procedure for implementing the momentum factor:

Step 1. A number of registers equal to the number of weights (w) in the network are designated for weight change ($\Delta w[n]$). All registers are initialized to zero before presenting the first pattern (p).

Step 2. First pattern is presented and weight changes, $\Delta w[n+1]$, are calculated using equations (4.57) and (4.58). Here the momentum term is zero due to Step 1.

Step 3. Next pattern is presented and weight changes are calculated using the results of step 2 in the momentum term.

Step 4. All patterns are presented one after the other and corresponding weight changes are calculated in a similar fashion.

Step 5. Weight changes calculated for the last pattern are dropped.

Step 6. Feedforward and then go to step 1 for the next iteration until the desired error or maximum number of iterations is reached.

In other words the weight change calculated for the previous pattern is used in the momentum term for calculating the weight change for the next pattern (except the first pattern), and the weight change calculated for the last pattern is not considered for this purpose (see flowchart in Appendix A).

4.4.1.2 Analysis of Kim's Technique

Kim's technique suffers from two problems:

1. All the registers are initialized to zero at the beginning of each iteration thereby discarding the last calculation of weight change in the previous iteration. Hence the weight changes calculated for the first pattern are without a momentum term.
2. The weight change calculated for one pattern is used in the momentum term for calculation of weight change for the next pattern.

The training results indicate the following response attributable to the above mentioned observations:

1. The convergence range is extremely restricted. Weight values for most of the processing elements tend to cross the limits ($-1.0 \leq w \leq +1.0$) and start increasing without bound.
2. The network takes a long time (too many iterations) to converge.

4.4.1.3 Redefining the Implementation of Momentum Factor

In order to achieve better results, the procedure for implementing the momentum factor is redefined as follows:

Step 1. A number of registers, equal to the number of patterns (p), are designated for weight changes (Δw_p) in each weight (w) in the network. All the registers are initialized ($\Delta w_p[n] = 0$) at the beginning of the training mode.

Step 2. All patterns are presented one by one and weight changes, $\Delta w_p[n+1]$, are calculated using equations (4.57) and (4.58). Here, the momentum term is zero for all the patterns for the first iteration ($n = 0$) only.

Step 3. In the next iteration ($n+1$), weight changes, $\Delta w_p[n+1]$, are calculated using the results of step 2 in the momentum term.

Step 4. Go to step 2 until the desired error or maximum number of iterations is reached.

In essence, the weight change is calculated separately for each pattern and it is used in the momentum term for calculating the next weight change for the same pattern. Moreover, no weight change is discarded.

4.4.2 Updating the Weights

The network weights are not updated after presenting each pattern. Rather, the weights are modified only after all input patterns have been presented. After each pattern

is presented, all weight changes are calculated, as usual, but these changes are not immediately applied. Instead, the changes for each weight are summed separately for all the input patterns, and the sum is applied to modify the weight after each iteration. That is,

$$\Delta w_{ij}[n+1] = \sum_{\mu=1}^{p} \Delta_{\mu} w_{ij}[n]. \tag{4.59}$$

Note that n represents the iteration number rather than the presentation number, since the weights are updated only once per iteration for all the patterns. The flowchart in Appendix A elaborates the above procedure in detail.

4.4.3 Varying the Learning Rate (η)

An intelligence is incorporated in the program to check the rate of convergence and apply the appropriate remedial measures. The learning rate (η) is varied in accordance with the performance index (total error for all patterns for each iteration). If an update results in reduced total error, η is multiplied by a factor $\phi > 1$ for the next iteration. If a step produces a network with a total error more than a few (typically 1-5) percent above the previous value, all changes to the weights are rejected and η is multiplied by a factor $\beta < 1$. α is set equal to zero and the step is repeated. α is reset to its original value only after execution of a successful step.

The rationale behind this maneuver is the idea that as long as the topography of the terrain is relatively uniform and the descent is relatively smooth, the memory implicit in α will aid convergence. If, however, a step results in a degradation of the performance of the system, then clearly the topography of the terrain demands a change in the direction of the optimization. Moreover, experience incorporated in the term in α will be misleading rather than beneficial. Hence α is set to zero so that memory from the previous steps is lost. α will assume a nonzero value only after the network has taken a step that reduces the total error.

4.5 Summary

The mathematical derivation of the stochastic function in backpropagation algorithm has been presented. A critical analysis of the backpropagation technique is given. Measures taken by Kim to improve the performance index of the DMNN are discussed and possible improvements are suggested to further improve the performance of the DMNN.

CHAPTER 5

DMNN ANALYSIS AND APPLICATIONS

5.1 Introduction

The DMNN developed by Kim [1] has been simulated using a software model. The simulated DMNN is applied to the benchmark and the character recognition problems and its results are compared to an ordinary ANN using a sigmoidal function. The improvements suggested in Chapter 4 are also implemented and their affect is observed. The difficulties and constraints encountered during the course of this research are discussed at the end.

5.2 Methodology

5.2.1 General Methodology

Three networks have been considered for application and analysis: the DMNN, the modified DMNN and the ANN. The DMNN and the ANN incorporate the algorithm discussed in Sections 4.4.1.3 and 4.4.2. The modified DMNN incorporates the technique discussed in Section 4.4.3. Each network is tested for three basic classification problems:

the XOR, data encoder, and pattern recognizer problems. The sampling data (patterns) for each problem is divided into the training data and the testing data. Each network is trained first with the set of training data. To study the learning pattern, networks are trained iteratively for a wide range of values for the learning rate (η) and the momentum factor (α). Once the training is completed, the test data is applied to the trained networks. Based on these results, the performance of the networks is evaluated.

Each pattern in the sampling data is an array of binary values. The number of neurons in the input layer of a network equals the number of elements in the pattern vector plus one. The additional input is required to provide the synaptic elements located in the upper layers of the network with a constant threshold input of 1.0. The number of neurons in the output layer is the same as the number of categories into which the sampling patterns are to be classified.

The network configuration defines, in matrix form, the interconnection layout of the processing elements in the network. A minimal network configuration for solving a particular problem can be defined as the network that converges to an error state equal to or below the predefined error threshold with a minimum number of neurons in the hidden layers and minimum number of hidden layers. Kim suggests that a two layer network (one hidden layer) with minimal network configuration is enough for solving most problems. He selects 0.0 and 1.0 to represent a logic 0 and a logic 1 respectively. Also 0.45 and 0.55 have been used as target values corresponding to ON and OFF thresholds at the output neurons. The above mentioned selections are incorporated in the three networks for solving the test cases. Input variables are selected in the following manner:

Learning Rate (η): η is varied from 0.1 to 1.0 with a step size of 0.1.

Momentum Factor (α): α is varied from 0.0 to 0.9 with a step size of 0.1. Since this value is changed for each η, a total of 100 different combinations for each problem are applied to the networks.

Synaptic Weights (w): The synaptic weights are generated randomly. To be able to compare and draw meaningful conclusions, results are averaged over 20 repetitions for each combination.

Maximum Sum-squared Error (ss-error): A value of 0.00001 is used for all the problems because the higher values (ss-error > 0.00001) do not sufficiently train the networks.

A comparison of the selection of input variables with those selected by Kim, is given in Table 5.1.

Table 5.1. Comparison of Input Variables.

Input Variable	Kim's Value [1]	This Research
Learning Rate η	0.2	0.1 to 1.0
Momentum Factor α	0.5	0.0 to 0.9
SS - Error	0.0001	0.00001
Maximum Iterations	30,000	30,000
Range of Final Weights	-1.0 to +1.0	-1.0 to +1.0
Network Configuration	minimal	minimal
Range of Weight Generation	0.0 to +1.0	0.0 to +0.5

5.2.2 Training and Classification

Each network is trained with a set of training and target patterns. Thereafter it is tested for classification using a set of test patterns. The synaptic weights are randomly generated once and updated continuously during the training session until the desired convergence is achieved. The training steps are listed below:

Step 1. Load the desired network configuration which defines the network according to the nature of the problem. Input the values of learning rate (η) and the momentum factor (α). Also input the values of phi (ϕ) and beta (β) in case of the modified network discussed in Section 4.4.3.

Step 2. Randomize all the synaptic weights between 0.0 and +0.5.

Step 3. Apply all the training and target patterns one by one. Perform feedforward operation and calculate the sum-squared error for the network.

Step 4. Apply all the training and target patterns one by one. Perform the feedback operation to propagate the error in a backward direction and modify the synaptic weights.

Step 5. Go to step 3 until the network converges and the sum-squared error value reaches below the threshold value of 0.00001. If the network oscillates or is stuck in a local minima, terminate the operation after 30,000 iterations.

Step 6. Save the final weights for testing the desired patterns.

Step 7. Check if the values of all synaptic weights are valid, *i.e.*, $-1.0 \leq w \leq 1.0$. In case of invalid weights, repeat steps 1 to 6.

Flowcharts elaborating the above procedure are given in Appendix A.

5.3 Benchmark Problems

The benchmark problems include an XOR problem and an 8x3 data encoder problem. An analysis of these problems is presented in the following two sections.

5.3.1 XOR Problem Solver

The XOR problem is a classic problem involving hidden units. This problem has invariably been applied to the neural networks to study their nonlinear behavior. The XOR problem is applied to each network with a network configuration of 3x2x1. The performance of the DMNN for $0.1 \leq \eta \leq 0.8$ and a momentum factor, $\alpha = 0.2$, is shown in Figure 5.1. A three dimensional view of the network's learning pattern is presented in Figure 5.2, where, $0.1 \leq \eta \leq 0.8$ and $0.0 \leq \alpha \leq 1.0$. The graphical representation identifies the values of η and α for which the network learns faster. Figure 5.2 shows that the best combination is for $0.6 \leq \eta \leq 0.7$ and $0.3 \leq \alpha \leq 0.4$ when the network learns below 2000 iterations. These values are useful for the future learning and testing of the network. The 3-D graph in Figure 5.2 also indicate the operational bounds of the DMNN.

The learning pattern for the ANN is presented in Figures 5.3 and 5.4. The pattern indicates several peaks and troughs showing an extremely divergent behavior. For comparison, similar values of η and α have been selected. The network generally learns faster with the increase in both the above mentioned variables. Figure 5.4 indicates that the best learning values are, η = 0.8 and α =0.9, when the iterations are below 2000. These are, however, only the troughs, as the learning does not follow a set pattern.

In case of the modified DMNN, the training starts with an initial value of learning rate (η), and momentum factor (α), set by the operator. The learning rate is modified throughout the training cycle. It is therefore impractical to draw any meaningful sketch of its learning behavior. It has been noticed that the modified DMNN would invariably learn faster if the training is initiated with the acceptable values of η and α. Moreover, due to its construction, the network would respond to a wider range of η and α.

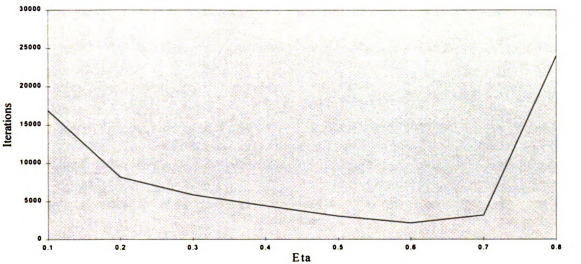


Figure 5.1. DMNN as XOR Solver for a Momentum Factor $\alpha = 0.2$.

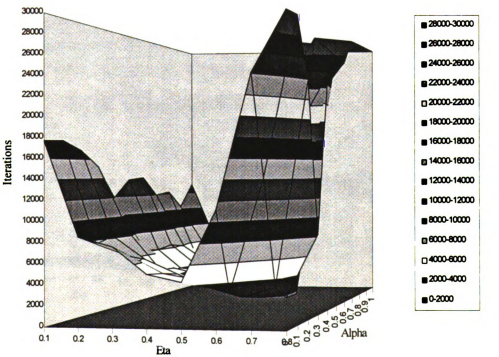


Figure 5.2. DMNN XOR Solver in 3-D, η vs α vs Number of Iterations.

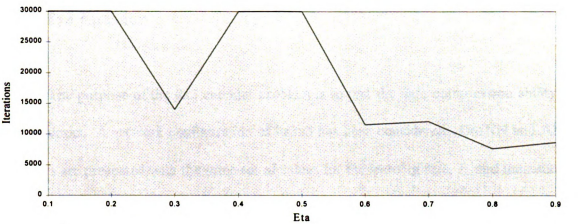


Figure 5.3. ANN as XOR Solver for a Momentum Factor $\alpha = 0.2$.

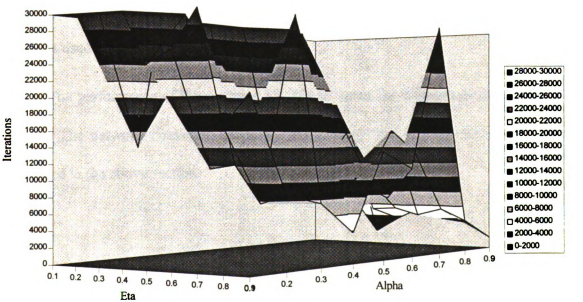


Figure 5.4. ANN XOR Solver in 3-D, η vs α vs Number of Iterations.

5.3.2 8x3 Encoder

The purpose of the 8x3 encoder problem is to test the data compression ability of the networks. A network configuration of 9x3x3 has been considered. DMNN and ANN encoders are presented with the same set of values for the learning rate, η, and momentum factor, α. The learning performance of the DMNN is represented graphically in Figures 5.5 and 5.6. The ANN counterpart is portrayed in Figures 5.7 and 5.8. The learning pattern shows that DMNN learns fast for $\eta \leq 0.3$. The performance degrades logarithmically as the learning rate is increased above 0.3. The ANN on the other hand, performs well for $\eta \geq 0.2$ and shows poor performance for lower values of η. The divergent behaviors of the two networks indicate different characteristics of the squashing functions used in the networks.

The performance of the modified DMNN is tested for different initial values of η and α. The network converges relatively faster when $\eta \leq 0.4$ for the same reasons mentioned in the above section.

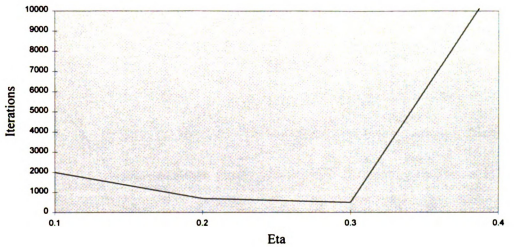


Figure 5.5. DMNN as 8x3 Encoder for a Momentum Factor $\alpha = 0.2$.

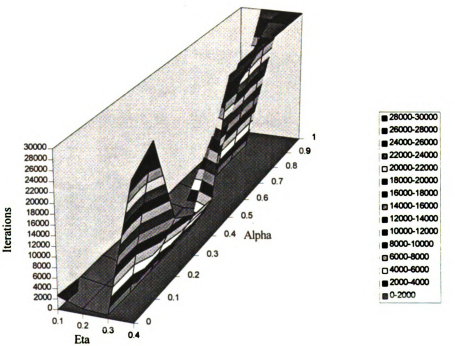


Figure 5.6. DMNN 8x3 Encoder in 3-D, η vs α vs Number of Iterations.

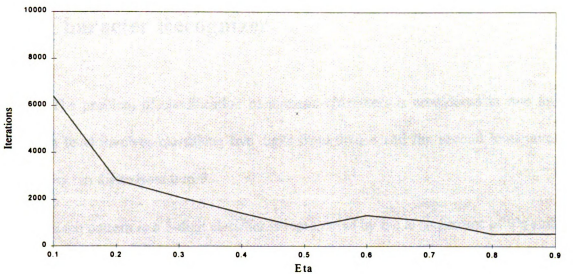


Figure 5.7. ANN as 8x3 Encoder for a Momentum Factor $\alpha = 0.2$.

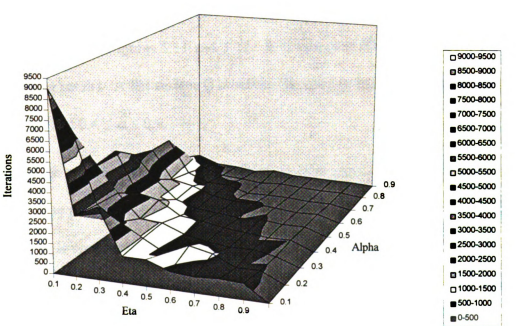


Figure 5.8. ANN 8x3 Encoder in 3-D, η vs α vs Number of Iterations.

5.4 Character Recognizer

The problem of classification of numeric characters is considered in two levels. The first level involves classifying five digits from 0 to 4 and the second level involves classifying ten digits from 0 to 9.

Each pattern in a 5-digit classifier is represented by the arrangement of 0's and 1's in a 6x4 bit matrix (Figure 3.13). The network configuration is 25x4x5. The network is trained first with a set of ideal digits (noise free). The simulation results are depicted by plots in Figures 5.9 and 5.10. The learning pattern indicates that the DMNN learns comparatively faster (less than 2000 iterations) for $\eta \geq 0.2$ and $0.1 \leq \alpha \leq 0.4$. The surface, however, looks rocky for $0.1 \leq \eta \leq 0.45$ and $0.4 \leq \alpha \leq 1.0$. The learning pattern for the ANN is shown in Figures 5.11 and 5.12. It is observed that the ANN produced more encouraging results. A flat surface is observed (learning in less than 1000 iterations) for $0.2 \leq \eta \leq 1.0$ and $0.4 \leq \alpha \leq 0.9$.

The 5-digit classifier is tested (after training on ideal digits) for a misclassification error on 10% noisy patterns and 20% noisy patterns. All networks showed similar results. The performance is considerably reduced when the test is conducted for 20% noisy data. The average misclassification error results are presented in Table 5.2. Five sets of noisy patterns, in each category, are considered for each network and the results are averaged out. The performance of the networks can improve considerably if they could also be

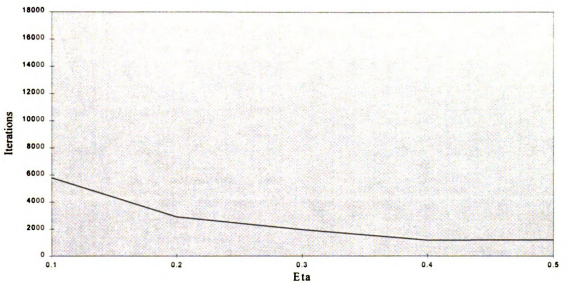


Figure 5.9. DMNN as 5-Digit Classifier for a Momentum Factor $\alpha = 0.2$.

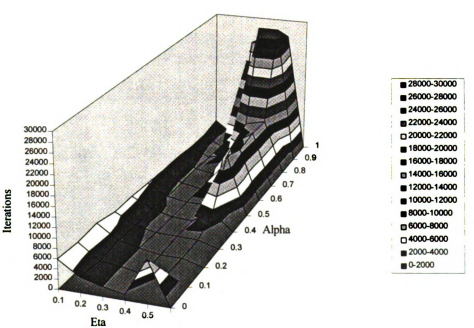


Figure 5.10. DMNN 5-Digit Classifier in 3-D, η vs α vs Number of Iterations.

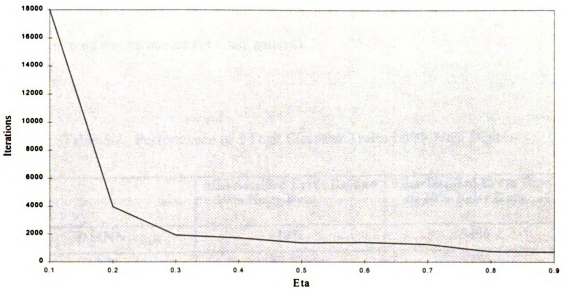


Figure 5.11. ANN as 5-Digit Classifier for a Momentum Factor $\alpha = 0.2$.

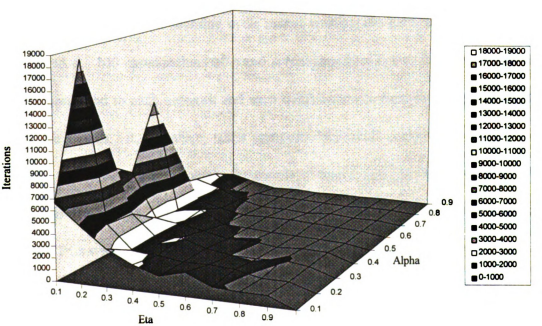


Figure 5.12. ANN 5-Digit Classifier in 3-D, η vs α vs Number of Iterations.

trained for noisy patterns. Due to time and machine constraints (Section 5.5), the networks could not be trained for noisy patterns.

Table 5.2. Performance of 5-Digit Classifier Trained with Ideal Digits.

	Missclassified Error Rate on 10% Noisy Data	Missclassified Error Rate on 20% Noisy Data
DMNN	18%	54%
ANN	17.5%	52%

It is observed that pattern classification is beyond the scope of the computational equipment used in this research. Even an ideal 5-digit classifier presents a computational load that takes a considerable time. Each learning cycle for an ideal 5-digit classifier takes an average of one hour. One learning cycle means training the network for a particular value of η and α. 100 combinations of η and α (ranging from 0.1 to 0.9 with a step size of 0.1) are presented to each network and each combination is repeated twenty times for averaging the results. It, therefore, takes approximately 2000 computational hours (84 days) for an 80486 based PC to obtain the results of one 5-digit DMNN classifier. The computational time for an ANN simulation is almost one half of the DMNN simulation. Eight similar PCs were engaged for 15 days to obtain the results of this experiment. The training with 10% noisy patterns was attempted but appreciable results could not be obtained within a reasonable time.

The second level involves classifying 10 digits from 0 to 9. The simulation entails a much larger data set: a pattern size of 5x7 bit matrix to represent each digit and a network configuration of 36x9x10. The network data segment becomes too large for a PC to handle. This experiment was, therefore, abandoned. A more powerful machine is the only solution to solve classification problems beyond an ideal 5-digit.

Input data in the form of network configuration, training patterns and test patterns for each problem discussed above, is given in Appendix B.

5.5 DMNN Simulation Analysis

The Digital Multilayer Neural Network was implemented using software simulation on a 80486 based PC with 8MB RAM. Kim [1] performed the simulation work on the Sun-SPARC stations, a much more powerful machine compared to a PC. This section analyses the simulation performance of Kim's DMNN on a PC. The need to incorporate improvements in the network is discussed and performance of different variations to the backpropagation algorithm is presented.

5.5.1 Kim's DMNN

The backpropagation algorithm variant used by Kim [1] is implemented first. The problems encountered during this phase are discussed in the following sections.

5.5.1.1 Program Compatibility

Kim developed a software simulation on Sun-SPARC workstations. Sun workstation operate under the UNIX operating system. The program is therefore not compatible with the DOS/windows environment. It was to be appropriately modified to work on a PC. In addition, many logical improvements were needed in the program to make it more efficient.

5.5.1.2 Invalid Synaptic Weights

Either the network would not converge, or it would generate a number of invalid synaptic weights ($w > \pm 1.0$). Once the synaptic weight values become invalid, they would increase without bound thus resulting in the overflow error.

5.5.1.3 Longer Convergence Cycles

In an attempt to relegate the problem of invalid weights, the values of learning rate (η) and momentum factor (α) are reduced in steps. The convergence starts at lower values of learning rate ($\eta < 0.2$), which makes the learning process very slow.

5.5.1.4 Implementation of Momentum Factor

Another aspect of Kim's network is that the effect of past weight change of one pattern is added in the momentum term for calculation of the weight change for the next

pattern, thus resulting in a longer convergence cycle. The details of the procedure for implementing the momentum factor are adequately covered in Section 4.4.1.1.

5.5.2 Variations and Improvements

In view of the above mentioned problems, there was a need to improve the performance of the program especially in the PC environment. A number of modifications are incorporated, both in the program structure and in the implementation technique, to make the network more efficient. The major modifications are discussed in the subsequent sections.

5.5.2.1 Program Compatibility

The program was made compatible with DOS for the PC. To compensate for the PC DOS's limitation of a 64K stack segment, most of the local variables are converted into global variables thus making them available via the 'heap'. All variables in the program are converted into double precision numbers (64-bit wide). The program is compiled with a Borland C++ (Ver. 3.1) compiler utilizing its options for speed optimization. To make most of the RAM available to the network, a DOS-Extender utility is employed. The PC is upgraded to a 486DX2-66MHz processor with an additional 8MB of RAM. In addition, eight PCs with similar capabilities are engaged in an effort to obtain the required results within a reasonable time.

5.5.2.2 Generation of Random Weights

Slight modifications were made in the pseudo random number generator. Instead of taking the seed from the user, a seed is now generated internally using the facility of Borland C. The new design is code efficient.

5.5.2.3 Redefining the Implementation of Momentum Factor

The momentum factor is implemented by allocating separate registers (equal to the number of patterns) for the weight change in each synaptic weights. The details of this procedure is narrated in Section 4.4.1.2. This modification appreciably reduced the number of iterations and improved the network learning.

5.5.2.4 Varying the Learning Rate (η)

The learning rate (η) is varied according to the performance index. Section 4.4.3 discusses the procedure for implementing this heuristic technique. A separate network (modified DMNN) is implemented and is subjected to the defined problems. The network converges much faster since the learning rate changes automatically, according to the topography of the error surface.

5.6 Summary

Three different networks: the DMNN, modified DMNN and ANN, simulated in C, are subjected to application problems. The networks are trained for a variety of learning rates (η) and momentum factors (α). The learning pattern of each network is drawn and its performance is compared/discussed. The software program used by Kim [1] is discussed. Various improvements, suggested in Chapter 4, are implemented and their effect is evaluated.

CHAPTER 6

CONCLUSION

A functional corroboration of the statistical model of DMNN developed by Kim [1] has been performed in this thesis. The statistical model is used in the feedforward multilayer neural network utilizing the backpropagation training algorithm. The stochastic function is compared with the standard sigmoidal function by subjecting the two networks to test problems. It has been found that while Kim's architecture has the desirable properties of high speed and high density on a chip, the simulated DMNN generally produces competitive results when compared with an ordinary ANN. An improved backpropagation algorithm is implemented in an effort to improve the training of the network. As a result a small scale pattern classification has been achieved.

This chapter summarizes the research work, identifies the major contributions made by this work and concludes with the outlining of possible future research tasks.

6.1 Summary

Presently, research is underway to develop a fast, space efficient and programmable architecture for dedicated VLSI ANNs which can be applied effectively to the real engineering problems. The development of a pulse mode digital multilayer neural

network by Kim [1] is a step towards the achievement of this goal. Kim developed a DMNN architecture based on stochastic computing techniques which can be implemented by using simple logic gates. Kim modeled the DMNN using VHDL code and analyzed its performance in data compression and character classification problems.

The aim of this research was to validate the results of Kim's model with a software simulation technique using the C programming language. The stochastic function developed by Kim [1] was employed in the backpropagation algorithm. A two layer DMNN was simulated using a C program. The backpropagation algorithm was analyzed, its weaknesses identified and various techniques were suggested for accelerating the convergence process.

The DMNN and its improved variants are subjected to the problems of XOR, data encoding (8x3) and character classification (5-digit and 10-digit). The simulation results indicate that the performance of the stochastic function is competitive, in this computing environment, with that of a sigmoidal function.

An 80486 DX2 processor based personal computer was used for the training simulations. Due to the intensive computational nature of the character classification problems, these could not be represented adequately. In an effort to solve these problems, the program structure and logic was improved appropriately.

To reduce the number of iterations required for the convergence, three modifications are proposed in the backpropagation algorithm:

1. The weight change, ($\Delta w[n]$), is calculated separately for each pattern and is used for calculation of the next weight change, ($\Delta w[n+1]$), for the same pattern. This method is more logical and has proved to reduce the training cycles.

2. Instead of wasting the effect of past weight change for each pattern, it is saved and used in the next iteration. This change not only proved to decrease the number of iterations but also assisted in reducing the number of invalid weight values.

3. The learning rate (η) and the momentum factor (α) are varied dynamically so that the system utilizes a near-optimum η, determined by the local optimization topography. Moreover, the momentum factor (α) is set to zero whenever its effect is misleading rather than being beneficial.

The DOS extender utility is used in an attempt to free maximum memory for the program. Eight systems are utilized to achieve the effect of parallel processing for the pattern classification problems.

As a result of these changes, the 5-digit pattern classifier has been successfully trained and tested for the ideal digits (noise free). The DMNN training involves computations which are generally beyond the capabilities of a personal computer. Even with all the possible efforts to enhance the capabilities of the PC, it was only possible to solve the 5-digit classification problem. When the number of processing elements are increased beyond a total of 34 and/or the number of patterns are more than five, the network is unable to converge within a reasonable time. Even with five patterns, a 5-digit classifier takes more than 8-10 hours for one training cycle. The misclassification error

shows that the network needs more training, in terms of number of patterns and in terms of the number of iterations, to improve its performance.

6.2 Contributions

The major contributions of this research are:

1. The DMNN based on the stochastic random processes, developed by Kim [1], has been simulated using C program and a functional validation of his results has been performed.

2. The same set of problems have been solved by an ANN using a sigmoidal function and the results of the two are compared.

3. Modifications to the backpropagation algorithm, incorporating the stochastic function, have been presented and their effectiveness has been confirmed.

6.3 Future Research

The DMNN proposed by Kim has been partially validated, using a software simulation technique, on a PC. A powerful system using multiprocessing can not only validate the results of a 10-digit character classifier, but it can further be used for solving more complex pattern classification problems. Furthermore, instead of C, an object-

oriented programming language (C++) can be applied which is better suited to such problems.

The backpropagation technique can be improved further by introducing the concept of "expected source values". Considering a more general form, the parameter adjustment between a node a and a node b can be given as

$$W_{ab}[n+1] = W_{ab}[n] + \eta \delta_b S_a + \alpha \Delta W_{ab}[n] \qquad (6.1)$$

where S_a, symbolizing source, is the output of node a or an external input to the network and δ_b is an associated error at the node b. T. Samad suggests that instead of using the current value of the source, S_a, use the following expected value of S_a

$$S_a = S_a + \beta \delta_a \qquad (6.2)$$

where β is any appropriate positive constant and δ_a is the error associated with node a. The modified learning rule has proven to be much faster [51].

Furthermore, recent advancements in fuzzy logic set theory [52,53] can be employed to vary η, β, and α based on variation in the output error and the training time (number of iterations) that has elapsed. For instance, the learning rate, η, can have following fuzzy rules:

1. $\eta(e,n)$ can be set to a large value when the output error, e, is large. This rule will indicate that weights are far away from the optimal value.

2. At other instances, $\eta(e,n)$ can be set to a small value when the output error, e, is small. Thus indicating that weights are getting closer or are in the neighborhood of optimal values.

3. $\eta(e,n)$ can be set to a large value if training time (number of iterations), is fairly small indicating the beginning of training cycle.

4. $\eta(e,n)$ can be set to a small value for better convergence if the training time (number of iterations) is significantly large indicating the final stages of the training process, and so forth.

Table 6.1 shows the possible fuzzy associative memory (FAM) rules for $\eta(e,n)$. Similar rules can be developed for other parameters, β and α. These rules are based on the relevant knowledge about the learning behavior.

Table 6.1. The Fuzzy Associative Memory (FAM) Rules for $\eta(e,n)$.

	Output Error			
Time (Iterations)	**Very small**	**Small**	**Large**	**Very large**
Less	*small*	*large*	*very large*	*very large*
Moderate	*very small*	*small*	*large*	*very large*
Large	*very small*	*small*	*large*	*large*

APPENDICES

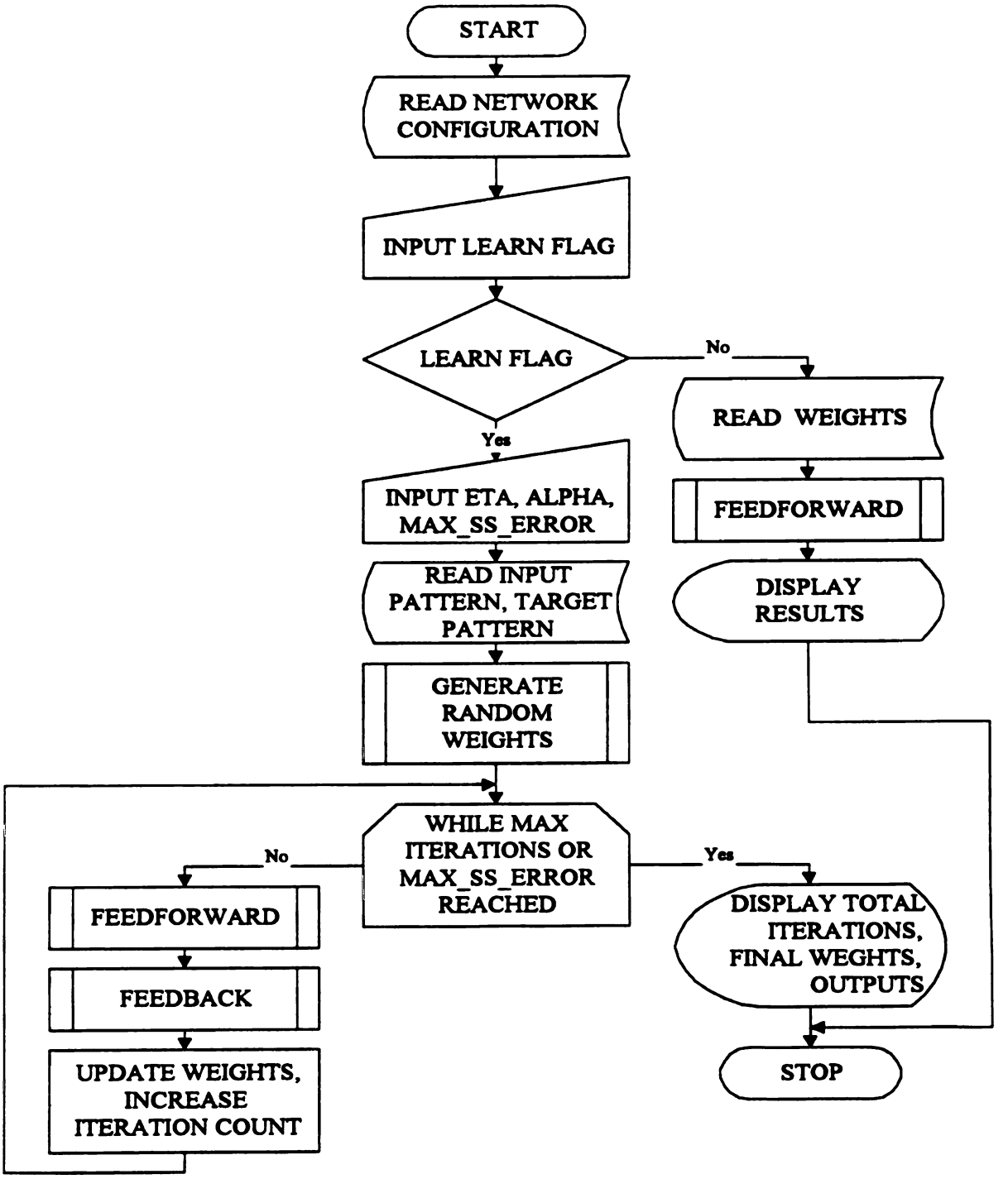
APPENDIX A

Flowcharts

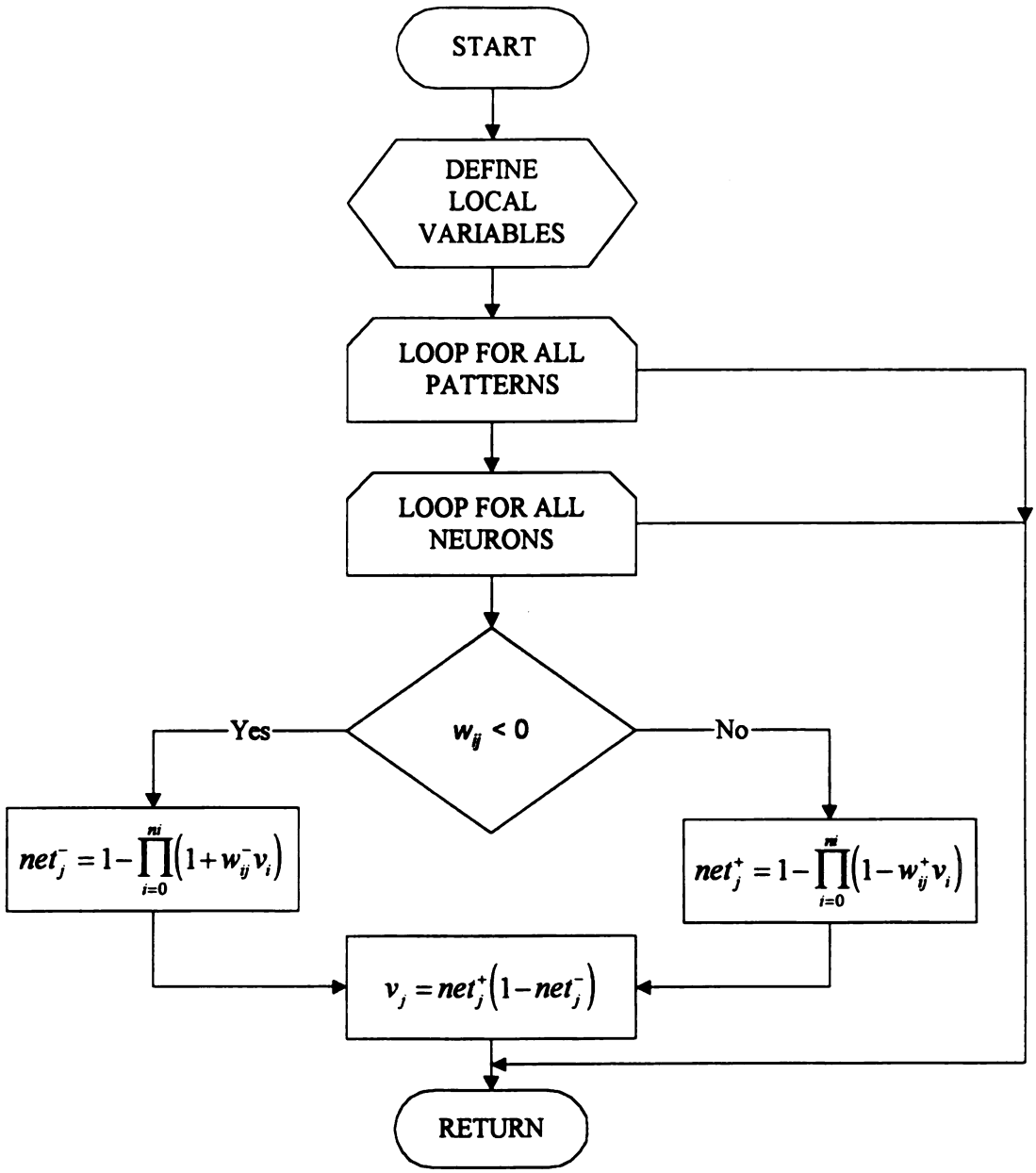
APPENDIX A

Flowcharts

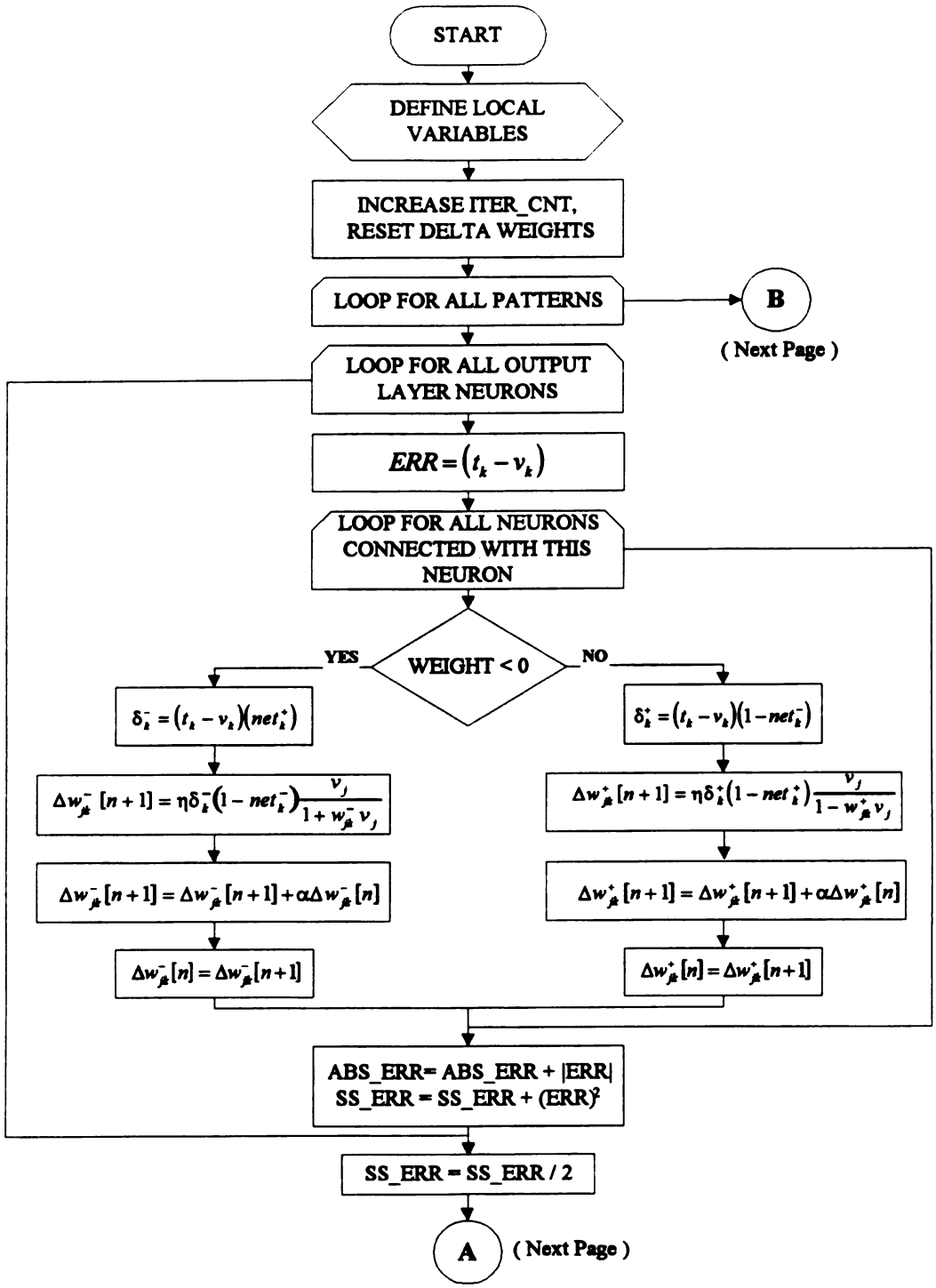
A.1 DMNN Operation

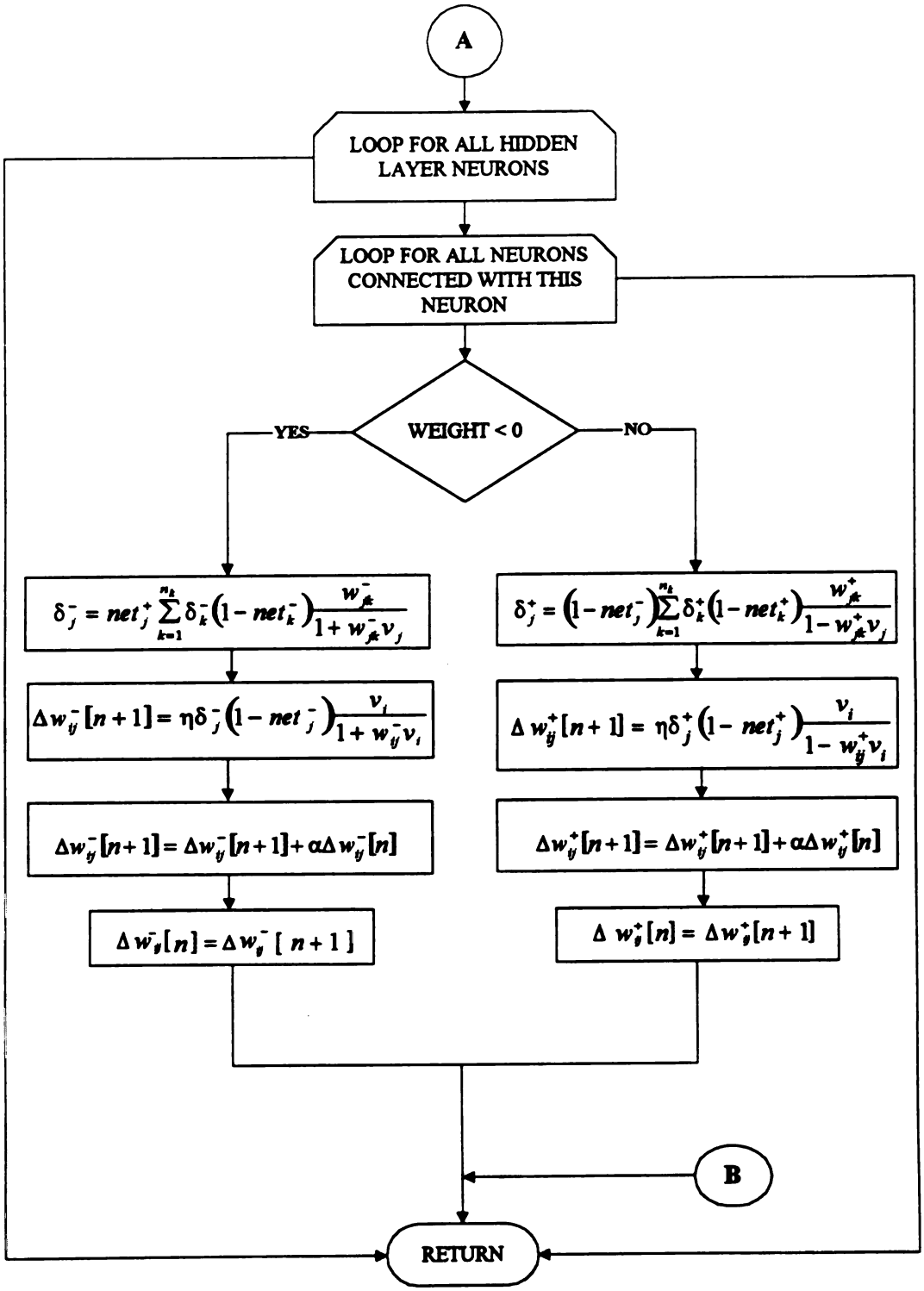


A.2 Feedforward Operation



A.3 Feedback Operation




A
LOOP FOR ALL HIDDEN LAYER NEURONS
LOOP FOR ALL NEURONS CONNECTED WITH THIS NEURON
WEIGHT < 0
YES
NO
$\delta_j^- = net_j^+ \sum_{k=1}^{n_k} \delta_k^- \left(1 - net_k^-\right) \frac{w_{jk}^-}{1 + w_{jk}^- v_j}$
$\Delta w_{ij}^-[n+1] = \eta \delta_j^- \left(1 - net_j^-\right) \frac{v_i}{1 + w_{ij}^- v_i}$
$\Delta w_{ij}^-[n+1] = \Delta w_{ij}^-[n+1] + \alpha \Delta w_{ij}^-[n]$
$\Delta w_{ij}^-[n] = \Delta w_{ij}^-[n+1]$
$\delta_j^+ = \left(1 - net_j^-\right) \sum_{k=1}^{n_k} \delta_k^+ \left(1 - net_k^+\right) \frac{w_{jk}^+}{1 - w_{jk}^+ v_j}$
$\Delta w_{ij}^+[n+1] = \eta \delta_j^+ \left(1 - net_j^+\right) \frac{v_i}{1 - w_{ij}^+ v_i}$
$\Delta w_{ij}^+[n+1] = \Delta w_{ij}^+[n+1] + \alpha \Delta w_{ij}^+[n]$
$\Delta w_{ij}^+[n] = \Delta w_{ij}^+[n+1]$
B
RETURN

APPENDIX B

Input Data for DMNN Binary Classifiers

APPENDIX B

Input Data for DMNN Binary Classifiers

Input data in the form of network configuration, training patterns and testing patterns used for XOR, 8x3 encoding and 5-digit classification problems are listed here. For network configuration, the value in row *i* and column *j* represents the connectivity between neurons *i* and *j* respectively. The *i*th element of the training and test pattern vectors is the value applied to the *i*th neuron in the input layer. Input data for the 10-digit classification problem has been omitted since the problem could not be represented. The data is available in [1].

B.1 XOR Problem

B.1.1 Network Configuration (xor.cnf)

1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	1
1	1	0	0	0	1
0	0	1	1	0	1
0	0	0	0	0	1

B.1.2 Training Patterns (xor.pat)

0.0	0.0	0.0	0.0	0.45	1.0
0.0	1.0	0.0	0.0	0.55	1.0
1.0	0.0	0.0	0.0	0.55	1.0
1.0	1.0	0.0	0.0	0.45	1.0

B.1.3 Test Patterns (xor.tgt)

0.0	0.0	0.0	0.0	0.0	1.0
0.0	1.0	0.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	0.0	1.0
1.0	1.0	0.0	0.0	0.0	1.0

B.2 8x3 Encoding Problem

B.2.1 Network Configuration (8x3.cnf)

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	1	1	0	0	0	1
0	0	0	0	0	0	0	0	1	1	1	0	0	0	1
0	0	0	0	0	0	0	0	1	1	1	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

B.2.2 Training Patterns (8x3.pat)

1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.45	0.45	0.45	1.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.45	0.45	0.55	1.0
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.45	0.55	0.45	1.0
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.45	0.55	0.55	1.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.55	0.45	0.45	1.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.55	0.45	0.55	1.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.55	0.55	0.45	1.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.55	0.55	0.55	1.0

B.2.3 Test Patterns (8x3.tgt)

1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

B.3 5-Digit Classification Problem

B.3.1 Network Configuration (5d.cnf)

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 1
```

B.3.2 Training Patterns for Ideal Digits (5d.pat)

```
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.45 0.55 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.55 0.45 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.55 0.45 0.45 1.0
1.0 1.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.55 0.45 0.45 0.45 1.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.55 0.45 0.45 0.45 0.45 1.0
```

B.3.3 Test Patterns for Ideal Digits (5d.tgt)

```
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

B.3.4 Test Patterns for Ideal Digits + 10% Noise (three noisy patterns per digit)

```
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 1.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 0.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 1.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0

B.3.5 Training Patterns for Ideal Digits + 10% Noise (three noisy patterns per digit)

0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.45 0.55 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.45 0.55 1.0
1.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.45 0.55 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 1.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.45 0.55 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.55 0.45 1.0
0.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.55 0.45 1.0
0.0 1.0 0.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 1.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.55 0.45 1.0
0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.45 0.55 0.45 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.55 0.45 0.45 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.55 0.45 0.45 1.0
0.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.55 0.45 0.45 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.45 0.45 0.55 0.45 0.45 1.0
1.0 1.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.55 0.45 0.45 0.45 1.0
1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.55 0.45 0.45 0.45 1.0
1.0 1.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.55 0.45 0.45 0.45 1.0
1.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.45 0.55 0.45 0.45 0.45 1.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.55 0.45 0.45 0.45 0.45 1.0
1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.55 0.45 0.45 0.45 0.45 1.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.55 0.45 0.45 0.45 0.45 1.0
1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.55 0.45 0.45 0.45 0.45 1.0

B.3.6 Test Patterns for Ideal Digits + 10% Noise (three noisy patterns per digit) + 20% Noise (three noisy patterns per digit)

0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 1.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0
1.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0
0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 1.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 0.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 1.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0
0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0
1.0 0.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 0.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 0.0 0.0 1.0
1.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0

1.0 1.0 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 1.0 0.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0
0.0 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0
0.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 0.0 1.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 1.0 1.0 0.0 1.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0
1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0
1.0 1.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0

LIST OF REFERENCES

LIST OF REFERENCES

[1] Y.C. Kim, "Architecture and Statistical Model of a Pulse Mode Digital Multilayer Neural Network," Ph. D. Dissertation, Michigan State University, 1993.

[2] R.P. Lipmann, "An Introduction to Computing with Neural Nets," IEEE ASSP Magazine, pp. 4-22, April 1987.

[3] D.E. Rumelhart, G.E. Hilton, and R.J. Williams, "Learning Internal Representations by Error Propagation," Parallel Distributed Processing, Cambridge, MA, MIT Press, Vol. 1, pp. 318-362, 1986.

[4] F.C. Hoppensteadt, "An Introduction to the Mathematics of Neurons," Cambridge University Press, 1986.

[5] C. Mead, M. Sivilotti, and M. Emerling, "A Novel Associative Memory Implemented using Collective Computation," Chapel Hill conference on VLSI, pp. 329-342, 1985.

[6] H.P. Graf, L.D. Jackel, W.E. Hubbard, "VLSI Implementation of a Neural Network Model," IEEE Computer, pp. 41-49, March 1988.

[7] H.P. Graf, L.D. Jackel, "Analog Electronic Neural Network Circuits," IEEE Circuits and Devices, pp. 44-49, July 1989.

[8] W. Wike, D.E. van den Bout and T.Miller III, "The VLSI Implementation of STONN," Proc. IJCNN, Vol. III, pp. 443-452, 1987.

[9] D.E. van den Bout and T.K. Miller, "A Digital Architecture employing Stochasticisim for the Simulation of Hopfield Neural Nets," IEEE Trans. on Circuits and Systems, Vol. 36(5), pp. 732-738, May 1989.

[10] M.S. Tomlinson Jr, D.J. Walker, M.A. Sivilotti, "A Digital Neural Network Architecture for VLSI," Proc. IJCNN, Vol. II, pp. 545-556, 1990.

[11] A. Moopenn, A.P. Thakoor, T. Duong, and S.K. Khanna, "A Neurocomputer based on an Analog-Digital Hybrid Architecture," Proc. IEEE ICNN, Vol. III, pp. 479-486, 1987.

[12] Y. Tsividis, and S. Satyanarayama, "Analogue Circuits for Variable Synapse Electronic Neural Networks," Electronic Letter, 1987.

[13] D. Nguyen and F. Holt, "Stochastic Processing in a Neural Network Application," Proc. IEEE ICNN, Vol. III, pp. 281-291, 1987.

[14] W. James, "Psychology: Briefer Course," Holt, New York, 1980.

[15] W.C. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," Bulletin of Mathematical Biophysics, Vol. 5, pp. 115-133, 1943.

[16] D.O. Hebb, "The Organization of Behavior," John Wiely, New York, NY, 1949.

[17] J.V. Neumann, "The Computer and the Brain," Yale University Press, 1958.

[18] F. Rosenblatt, "The Perceptron: A Probablistic Model for Information Storage and Organization in the Brain," Psychological Review, Vol. 65, pp. 386-408, 1958.

[19] B. Widrow and M.E. Hoff, "Adaptive Switching Circuits," 1960 IRE WESCON Convention Record, Part 4, Computers:, Man-Machine Systems, Los Angles, pp. 96-104, 1940.

[20] M. Minsky and S. Papert, "Perceptron," MIT Press, Cambridge, MA, 1969.

[21] J.A. Anderson, "A Simple Neural Network Generating on Interactive Memory," Mathematical Biosciences, Vol. 14, pp. 197-220, 1972.

[22] K. Fukushima, "Neocognitron: A Self-Organizing Neural Network for a Mechanism of Pattern Recognition Unaffected by a Shift in Position," Biological Cybernetics, pp. 36-193, 1980.

[23] T. Kohonen, "Associative Memory-A System Theoretical Approach," Springer Press, New York, 1977.

[24] J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," Proc. Natl. Acad. Sci., Vol. 79, pp. 2554-2558, 1982.

[25] J.J. Hopfield, "Neurons with Graded Response have Collective Computational Properties like Those of Two-State Neurons," Proc. Natl. Acad. Sci., Vol. 81, pp. 3088-3092, 1984.

[26] C. Mead, "Analog VLSI and Neural Systems," Addison-Wesley, MA, 1989.

[27] J. Darnell, H. Lodish and D. Baltimore, "Molecular Cell Biology," Scientific American Books, New York, pp. 715-765, 1986.

[28] B. Widrow, and S. Stearns, "Adaptive Signal Processing," Pren. Hall, 1985.

[29] T. Kohonen, "Self Organization and Associative Memory," Springer-Verlag, Second Edition, Berlin, 1987.

[30] D.W. Tank and J.J. Hopfield, "Simple Neural Optimization Neural Networks: An A/D Converter, Signal Decision Circuit, and Linear Programming Circuits," IEEE Trans. on Circuits and Systems, Vol. CAS-33, No. 5, pp. 533-541, May 1986.

[31] D.W. Tank and J.J. Hopfield, "Collective Computation in Neuronlike Circuits," Scientific American, pp. 104-114, Dec. 1987.

[32] F. Rosenblatt, "Principles of Neurodynamics," New York, Spartan, 1962.

[33] A. Guez and Z. Ahmad, "Solution to the Inverse Kinematics Problem in Robotics by Neural Networks," Proc. IJCNN, Vol. II, pp. 617-624, 1988.

[34] M. Kuperstein and J. Wang, "Neural Controller for Adaptive Movements with Unforeseen Payloads," IEEE Trans. on Neural Networks, Vol. 1, No. 1, March 1990.

[35] G.E. Hinton and T.J. Sejnowski, "Optimal Perceptual Inference," Proc. IEEE Conference on Computer Vision and Pattern Recognition, Washington, pp. 448-453, 1983.

[36] G.E. Hinton and T.J. Sejnowski, "Learning and Relearning in Boltzmann Machines," Parallel Distributed Procesing, Vol. III, pp. 721-734, 1987.

[37] Y. Tsividis and S. Satyanarayama, "Analog Circuits for Variable-Synapse Electronic Neural Networks," Electronic Letters, Vol. 23, pp. 1312-1313, 1987.

[38] Y. Suzuki and L. Atlas, "A Study of Regular Architecture for Digital Implementation of Neural Networks," Proc. IEEE Int. Symposium on Circuits and Systems, Portland, May 1989.

[39] Y. Suzuki and L. Atlas, "A Comparison of Processor Topologies for a Fast Trainable Neural Network for Speech Recognition," Proc. IEEE Int. Conf. on Acoustic, Speech, and Signal Processing, Glassgow, May 1989.

[40] D.A. Pomerleau, G.L. Gusciora, D.S. Touretzky, H.T. Kung, "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second," Proc. IJCNN, Vol. II, pp. 143-150, 1988.

[41] B.M. Forrest, D. Roweth, N. Stround, D.J. Wallace, and G.V. Wilson, "Implementing Neural Network Models on Parallel Computers," Computer Journal, Vol. 30(5), pp. 32-41, 1989.

[42] R. Kuczewsk, M. Myers, and W. Crawford, "Neurocomputer Workstations and Processors: Approaches and Applications," Proc. IJCNN, Vol. III, pp. 487-500, 1988.

[43] Y.C. Kim and M.A. Shanblatt, "An Implementable Digital Multilayer Neural Network (DMNN)," Proc. IJCNN, Vol. II, pp. 594-600, 1992.

[44] B.R. Gaines, "Stochastic Computing," Spring Joint Computer Conf., Vol. 30, pp. 149-156, 1966.

[45] S.T. Ribeiro, "Random Pulse Machine," IEEE Trans. on Electronic Computers, Col. EC-16, No. 3, pp. 261-276, June 1967.

[46] A Design & Test, "Behavioral Description Languages," IEEE Design & Test of Computer, pp. 56-58, 1990.

[47] C.K. Keshavachandra and M.A. Shanblatt, "Modelling Artificial Neural Networks using VHDL," A technical Report, MSU-ENGR-90-009, 1990.

[48] B. Widrow and S. Stearns, "Adaptive Signal Processing," Prentice-Hall, 1985.

[49] M. Jurik, "Back Error Propagation: A Critique," Proc., IEEE, May 1988.

[50] T.P. Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, "Accelerating the Convergence of the Backpropagation Method," Biol. Cybern. 59, pp. 257-263, 1988.

[51] T. Samad, "Backpropagation with Expected Source Values," Neural Networks, Vol. 4, pp. 615-618, 1991.

[52] C.C. Lee, "Fuzzy Logic in Control Systems: Fuzzy Logic Controller-Part I," IEEE Trans. on Systems, Man and Cyber. Vol. 20, No. 2, pp. 404-418, March/April 1990.

[53] C.C. Lee, "Fuzzy Logic in Control Systems: Fuzzy Logic Controller-Part II," IEEE Trans. on Systems, Man and Cyber. Vol. 20, No. 2, pp. 419-432, March/April 1990.