This is to certify that the

dissertation entitled

Architecture and Statistical Model of a Pulse-Mode Digital Multilayer Neural Network

presented by

Young-Chul Kim

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Electrical Engineering

Michael Shanblatt
Major professor

Date Feb. 9, 1993

ARCHITECTURE AND STATISTICAL MODEL OF A PULSE-MODE DIGITAL MULTILAYER NEURAL NETWORK

By

Young-Chul Kim

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1993

ABSTRACT

ARCHITECTURE AND STATISTICAL MODEL OF A PULSE-MODE DIGITAL MULTILAYER NEURAL NETWORK

By

Young-Chul Kim

A new architecture for a pulse-mode digital neural network is presented. Algebraic neural operations are replaced by stochastic processes using pseudo-random pulse sequences. Synaptic weights and neuron states are represented as probabilities and estimated as average rates of pulse occurrences in corresponding pulse sequences. A statistical model of error (or noise) is developed to estimate relative accuracy associated with stochastic computing in terms of a mean and a variance.

The stochastic computing model translates into simple logic gates as basic computing elements leading to a high neuron-density on a chip. Furthermore, the use of simple logic gates for neural operations, the pulse-mode signal representation, and the modular design techniques lead to a massively parallel yet compact and flexible network architecture well-suited for VLSI implementation. Any size feed-forward network can be configured using the modules. Processing speed is independent of the network size.

Multilayer feed-forward networks are modeled and applied to pattern classification problems such as encoding and character recognition. The architecture and all digital sub-components in the proposed neural network are modeled and simulated in VHDL. Computational accuracy is analyzed and the network performance is evaluated in terms of a correct classification rate. The simulation experiments in these applications show the network performance is competitive with that of deterministic DMNN simulations and ordinary back-propagation networks while retaining the desirable properties of high speed and high density on a chip.

To my parents and my wife

ACKNOWLEDGEMENTS

I would like to thank my major advisor, Dr. Michael A. Shanblatt, for his guidance and encouragement throughout the years of this research.

I also want to thank all the members of my Ph.D guidance committee, Dr. P. David Fisher, Dr. Chin-Long Wey, Dr. Moon-Jung Chung, and Dr. Jacob Plotkin, for their valuable comments and suggestions.

Finally, I wish to dedicate this dissertation to my parents for their love, understanding, and support, my wife, Gyea-Sook Kim, for her love, patience, encouragement, and my lovely children, Jong-Seok and So-Youn.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Artificial neural networks (ANN) present a practical approach to solving computationally intensive and (or) ill-defined problems such as pattern recognition, optimization, adaptive control, associative memory, and some complex information processing tasks. Dedicated VLSI implementation is crucial to building fast ANNs fully utilizing the parallelism embedded in ANN computations. This dissertation presents a new architecture for a digital feedforward neural network using stochastic computing techniques. Random noise effects in this architecture are also presented. The applicability of the network is demonstrated using pattern classification examples. This includes the network architecture, analysis, modeling, simulation, and applications. This chapter begins with a brief overview of the ANN implementation models. The problem to be solved is then defined, followed by the research tasks. Finally, the organization of this dissertation is outlined.

1.1 Overview

An artificial neural network is a highly interconnected array of simple computing elements inspired by the computational strengths of biological neural systems. The structure of individual nerve cells, called *neurons*, in biological neural systems is well understood. The neuron is specialized to conduct electrochemical impulses from or to sensory organs and other neurons. Its function is accomplished by means of hairlike nerve fibers. However, it is not yet well known how this neural network with its massive parallel interconnections functions as memory and manipulates complex human behaviors. Over the last decade many researchers from the fields of physics, mathematics, computer science, and engineering have provided useful theoretical analyses for various models of ANNs [1-11]. Neural network topologies and some design procedures have been proposed and many of these ANN models have been proven to be superior to conventional digital computers in areas such as pattern recognition, combinatorial optimization, associative memory, and human information processing tasks.

It is now widely believed that the massive parallelism and computational power of the human brain results from the global and complex interconnections among a large number of neurons rather than from the complexity of individual neurons. One of the major goals in the field of ANN implementation is to produce dedicated hardware that mimics those dense interconnections among a large number of neural elements. Most of the current ANN models, however, rely on computer simulations. With the help of the current advancements in integrated electronics, optical, and electro-optical technologies, dedicated hardware implementation of ANNs is now progressing [12-18]. To date, many analog and hybrid ANNs have been built using CMOS [12,15-17] or CCD technology [14]. Most of these are analog implementations of simple feedback

or feedforward neural networks. Analog implementation offers high-speed with low hardware cost. The primary disadvantages of analog processing are the inaccuracy of analog computations and the low design flexibility due to the physical constraints of analog electronic devices.

Digital ANN implementation can take advantage of some of the benefits of current VLSI technology such as well-understood and advanced design techniques and tools. 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 loop-up table (or memory) [19-23]. This approach has an increased area requirement and the level of parallelism decreases significantly due to the communication overhead. Recently, however, a new digital approach has been introduced to reduce the hardware requirement and to increase the level of parallelism. In this new approach, a synaptic multiplication and (or) a neuron activation function is implemented with simple logic gates using stochastic computing techniques [28-32].

In this dissertation, a set of fundamental research tasks are described which are aimed toward developing an efficient architecture and statistical model of a pulse-mode Digital Multilayer Neural Network (DMNN) based on stochastic computing. A statistical model is developed by which the accuracy of stochastic computing in the DMNN is analyzed. The operational characteristics and performance of the DMNN are quantified. The applicability of the developed network is demonstrated using benchmark comparisons and example character recognition problems. The results of this research contribute to the establishment of a pulse-mode DMNN which has a compact, flexible, and expandable structure.

1.2 Problem Statement

Many current ANN models rely on software simulations using serial or parallel digital computers. The speed of all software simulators, even those run on parallel machines, is far from equaling that of specialized VLSI ANNs. This is due mainly to the sequential nature of control flow and the communication overhead in digital computers. Some VLSI analog or hybrid ANN implementations have been built using matrices of fixed or variable resistors and nonlinear amplifiers [12,15-17,24,25,56]. Analog implementations of ANNs have the potential for high density; however, with current VLSI technology, it is very difficult to build large (or multichip) analog ANNs. This is mainly due to the inaccuracy of analog elements, the unavailability of reliable permanent analog storage devices, and design parameter variations such as noise, temperature, and high parasitic capacitances on external I/O pins. Difficulties in VLSI analog implementation of ANNs limit their density on a chip and constrained their applications, in turn, leading to a limitation in solving real engineering problems.

A digital approach is a viable alternative alleviating some of the above drawbacks to analog implementation. Digital implementation can take advantage of some of the benefits of current VLSI technology such as well-understood and advanced design techniques. Nevertheless, dedicated VLSI digital implementation has been less developed because a conventional digital approach to ANN implementation has an increased area requirement and complex connectivity. In order to build a large digital neural network, a space-efficient network architecture must be developed.

Some digital ANN architectures using stochastic computing techniques show the possibility of the low-cost and high-speed digital ANN implementation [28-32]. In these architectures, algebraic operations are replaced by random processes using ran-

dom pulse sequences. Simple logic gates combined with some other simplistic components perform multiplications and nonlinear transformation of signals. In this approach, 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. However, the overall feedforward network architecture which is programmable and expandable to any size has not yet been established. The mathematical model of the pulse-mode digital neural network also must be developed to estimate the relative accuracy of stochastic computations and to anticipate the network performance. Furthermore, the applicability of the developed neural network architecture must be verified using real-world application examples.

1.3 Research Tasks

The tasks of this research are to (1) develop a pulse-mode digital neuron architecture and the corresponding statistical model; (2) develop an efficient DMNN architecture and the statistical model of error (or noise) in the DMNN and analyze the accuracy of stochastic computations utilized in the DMNN; (3) Formulate the framework of VHDL modeling techniques for the DMNN and simulate the DMNN in VHDL; and (4) apply pattern classification problems to the DMNN and evaluate the network performance and compare the performance of the DMNN classifier with the results from other deterministic feedforward neural networks.

To develop a pulse-mode digital neuron model, the first step is to investigate various stochastic computing techniques using similarities between boolean algebra and probability algebra. The study is concentrated on developing a digital neuron model

in which a non-linear transfer (sigmoid) function is embedded, which is essential to ANN models. Also various digital neuron architectures are studied, including those published recently. Simultaneously, all necessary components are developed in such a way that each of them can contribute to a simple and regular neuron architecture. A statistical model of the neuron is developed. The computing accuracy of a synaptic multiplication and a neuron activation is estimated in terms of means and variances. A regular neuron architecture is sought in such a way that it leads to an expandable network architecture.

The second task is the development of a DMNN architecture and an analysis of the network. Existing feedforward network models with advanced architectures are explored. A flexible and modular architecture for the DMNN is sought such that the network can be programmed for different network configurations by simply connecting basic modules. The effective network structure to minimize the correlation between multiple pseudo-random pulse sequences is sought. The number of clock cycles per sampling period for the pulse-code representation of signals is determined in such a way that the required accuracy for a particular application problem is satisfied. A variable register length is one of the design issues for the DMNN. This decision may be made using knowledge gleaned from simulation results of actual problems. Simultaneously, network analysis is performed based on the statistical models to estimate the differences between the results obtained by the DMNN and those obtained by the deterministic calculation. At this stage, some assumptions are made for the analysis on distributions of synaptic weights and neuron activations because they depend highly on network architectures and application problems.

The third task involves VHDL modeling and simulation which demonstrate efficient behavioral modeling techniques for the DMNN. First of all, the clever use of VHDL semantics is necessary to get a precise model. Detailed investigations are undertaken on process statements, functions, and delay characteristics. A logic block

can be modeled using process statements and accompanying wait statements for the flow control in a VHDL description. In VHDL, a function subprogram defines an algorithm for computing values or representing the behavior of a hardware model. One of the useful functions in modeling digital neural networks is the bus resolution function which defines the resolution of output values for a common output signal. The delay model in VHDL must provide an accurate view of the timing associated with the logic gate. In addition, an effective naming convention is considered in order to develop VHDL models conveniently and to document them properly.

For the last task, some testbench problems and character recognition problems are applied to the developed DMNN. This demonstrates the applicability of the DMNN. Traditional pattern recognition systems rely on programmable algorithms based on statistical or syntactical approaches. They perform a mapping from the observation space to the interpretation space by extracting features from observed data and classifying the collected features into certain categories. The developed DMNN should self-organize the complex mapping required to solve the problem and provide a fast classification rate. The back-propagation algorithm for the DMNN is programmed in C and the DMNN will be modeled in VHDL. The network is trained on a host computer. After the training, the network configuration is determined and the classification of test patterns is performed by the DMNN. Testbench problems are tested on the DMNN at first. These problems include "exclusive OR" and "encoding" problems. The experimental results show the strength as well as the limitations of the DMNN. The performance measures include the number of classifications per second and the correct classification rate. Next, character classification problems are applied to demonstrate its applicability to real-world problems. The experimental results are compared with those of other approaches. For a particular problem, the proper representation for input and output patterns, and the best choice of a register length

in the DMNN is determined. As a result, a design procedure for a DMNN binary classifier is proposed.

1.4 Organization of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 contains the background discussion of related topics. It begins with a discussion of various artificial neural network models, and it briefly describes the existing hardware implementations of ANNs. This is followed by a discussion of traditional and ANN approaches for solving pattern classification problems. It ends with the brief discussion of VHDL characteristics and behavioral modeling techniques.

Chapter 3 presents the fundamentals of stochastic computing techniques. The techniques for generating random pulse sequences using Linear Feedback Shift Registers (LFSRs) and the randomness properties of the pulse sequences are presented. Synaptic weights and neuron activations are represented as generating probabilities with the pulse sequences. The statistical model of the generating probability is developed in terms of mean and variance. Then stochastic computing techniques to perform a synaptic multiplication and a signal intergration are discussed.

Chapter 4 proposes an architecture for the DMNN. The DMNN consists of synaptic elements, neuron body elements, and necessary connections. To develop an overall network architecture, modular design techniques are used. The DMNN is trained with the Back-Propagation (BP) learning rule suitable for the pulse-mode feed-forward neural network. A generic architecture of the DMNN coprocessor which can be attached to a host computer is proposed. The DMNN coprocessor is composed of the DMNN, a control unit, a memory unit, and some digital components. The network configuration for solving a particular problem is determined during the training ses-

sion. Once the training is completed, the determined synaptic weights and network configuration are loaded into the memory in the DMNN coprocessor from a host computer. Then, the programmed or hardwired control unit can be used to control the operations of the coprocessor during the classification session.

Random noises (errors) are involved in the stochastic computations of network operations. In Chapter 5, the statistical models of the network operations performed using stochastic computing techniques are presented. The relationship between the computing accuracy and the register length (or the sampling period), and the relationship between the computing accuracy and the network architecture will be discovered. The overall random noise effects on hidden and output layers are analyzed. The validity of the developed models and analysis results is justified by simulations.

In Chapter 6, the DMNN coprocessor is modeled and simulated in VHDL. Some testbench problems and character classification problems are applied to the coprocessor. A design procedure for solving binary classification problems with the DMNN coprocessor is proposed. Testbench problems are tested to see the applicability of the DMNN to binary classification problems. Network performance of DMNN character classifiers is evaluated in terms of successful classification rates. The network performance is compared with that of deterministic DMNN simulations or other ordinary back-propagation networks.

Finally, Chapter 7 contains the conclusions, contributions, and future direction of this work.

CHAPTER 2

Background

Many artificial neural network models have been developed based on current knowledge of biological neurons and with the help of available analytic methods for linear or nonlinear dynamic systems. Network topology, computational characteristics of neuron elements, and learning rules play key roles in specifying artificial neural networks. In the first section, feedback , feedforward, and recurrent models are discussed with learning rules associated with the network models. In recent years, many software and hardware implementations of these models have been developed. Among them, some software simulators, analog, and digital electronic ANNs are discussed in the next section, followed by related issues. Pattern classification is one of the major applications for feedforward ANNs. Traditional and neural network approaches used for pattern classification are presented. This chapter concludes with a brief discussion of behavioral modeling and VHSIC (Very High Speed Intergrated Circuit) Hardware Description Language (VHDL).

2.1 Artificial Neural Networks

In this section, a brief review of biological and artificial neurons is provided. Next, typical feedback network models such as the Hopfield model and the Kennedy-Chua model are discussed in relation to ANNs. This is followed by a discussion of feedforward network models and the Boltzmann machine as a recurrent network model.

2.1.1 Biological/Artificial Neurons

2.1.1.1 Biological Neuron

The biological nervous system consists of two principal classes of cells, the neurons and the neuroglia. The neuroglia are cells that fill the spaces between the neurons [33]. The neuron is a fundamental processing unit of all nervous systems. Most neurons contain four distinct regions which carry out the specialized functions of the cell: the cell body, the dendrites, the axon, and the synapse (Figure 2.1).

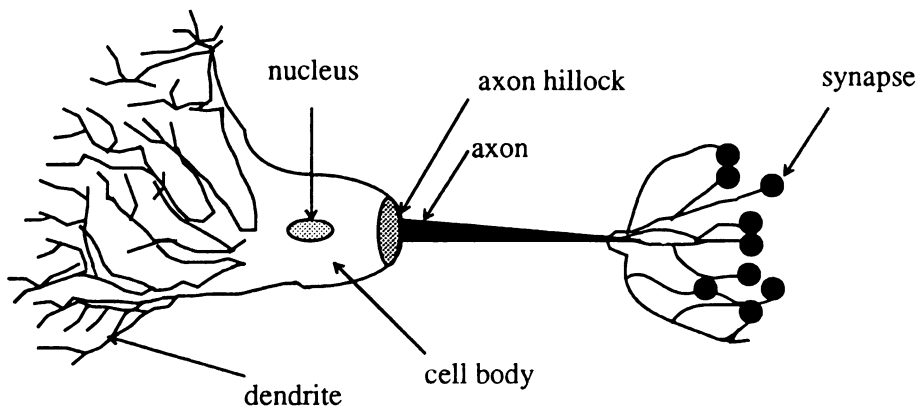


Figure 2.1. A biological neuron.

Axons are specialized for carrying information toward other cells without reducing the magnitude of signals. Action potentials originate at the axon hillock and travel to synapses, from which point signals are passed 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. If the electrical impulses are greater than a certain threshold, action potentials are generated and are actively conducted down the axon. The action potentials are pulse streams with a pulse-width of about 1 msec.

Synapses generally pass signals to other cells in only one direction; an 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 exist in biological neural systems: electrical and chemical. They differ in both structure and function. Cells communicating by electrical synapses are connected by gap junctions (Figure 2.2). This allows an electrical pulse to pass from the presynaptic cell to the postsynaptic cell. In chemical synapses, chemical substances, called neurotransmitters, are involved in passing the signals [33]. An action potential is generated in the postsynaptic cell.

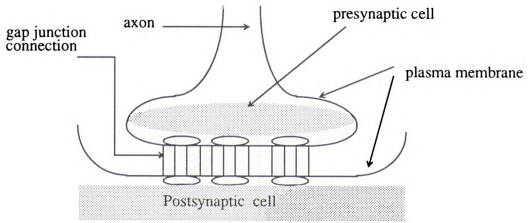


Figure 2.2. An electrical synapse.

Two types of signals occur in synapses: excitatory and inhibitory. With an excitatory synapse, the signal from the presynaptic cell causes a change in the plasma membrane of the postsynaptic cell that tends to induce an action potential. However, with an inhibitory synapse a nerve impulse in a presynaptic neuron affects the electrical properties of the postsynaptic membrane in such a way as to prevent the generation of an action potential. Excitatory and inhibitory stimuli often affect a single neuron in combination.

2.1.1.2 Artificial Neuron

An artificial neuron can be considered as a simple processing element which sums the weighted inputs and passes the result through a threshold or activation function. Figure 2.3 shows this simplified neuron.

The input signals, which come from either sensors or outputs of other neurons, form the input vector, $X = (x_1, \cdots, x_j, \cdots, x_n)$. The weights associated with each input form the weight vector, $W_i = (w_{i1}, \cdots, w_{ij}, \cdots, w_{in})$ for the ith neuron, where w_{ij} represents the connection strength between the ith and jth neurons. A threshold function can be modeled by associating a threshold θ_i in each neuron.

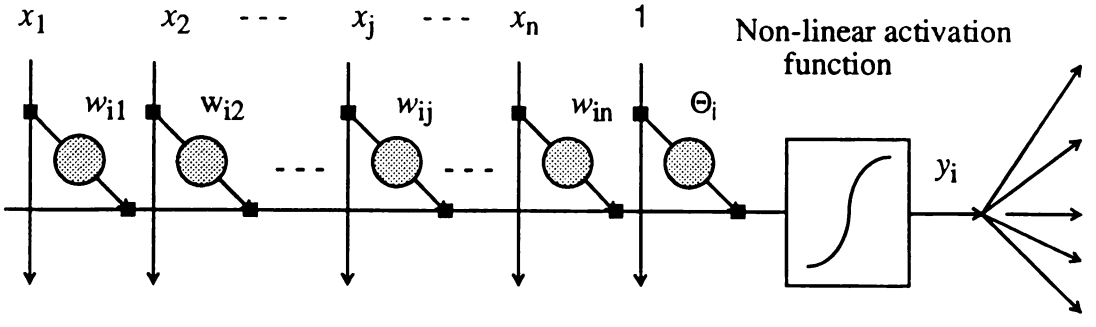


Figure 2.3. A simplified artificial neuron.

The output of the ith neuron, y_i, is then given by

$$y_i = f(X \cdot W_i - \theta_i) \tag{2.1}$$

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

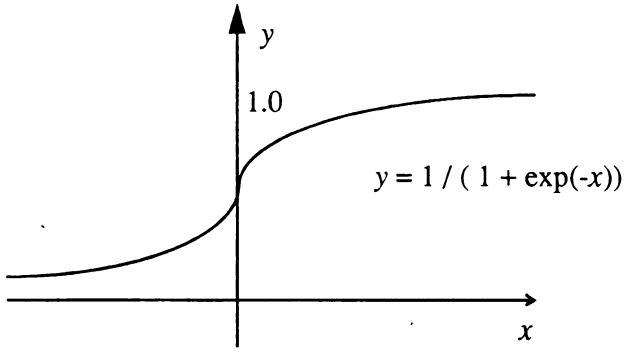


Figure 2.4. A sigmoid threshold function.

2.1.2 Feedback Model

Two feedback ANN models are reviewed: the Hopfield model and the Kennedy-Chua model. In feedback neural networks, neural elements are connected to one another by feedback paths from outputs to inputs of neural elements. Continuous-valued neural elements are normally implemented as electrical circuits, and the network dynamics are described by differential equations. A key issue of these networks is to define an energy function which always decreases during the dynamical evolution.

2.1.2.1 The Hopfield Model

The Hopfield model is a one-layer feedback network which consists of interconnected nonlinear analog neurons. Many implementations have been built based on this model. The general structure of this network is shown in Figure 2.5. In this model, each neuron is an amplifier with a capacitor C_i and a register ρ_i at the input node. The output of neuron j, v_j, is connected to the input of neuron i, u_i, via a conductance w_{ij}.

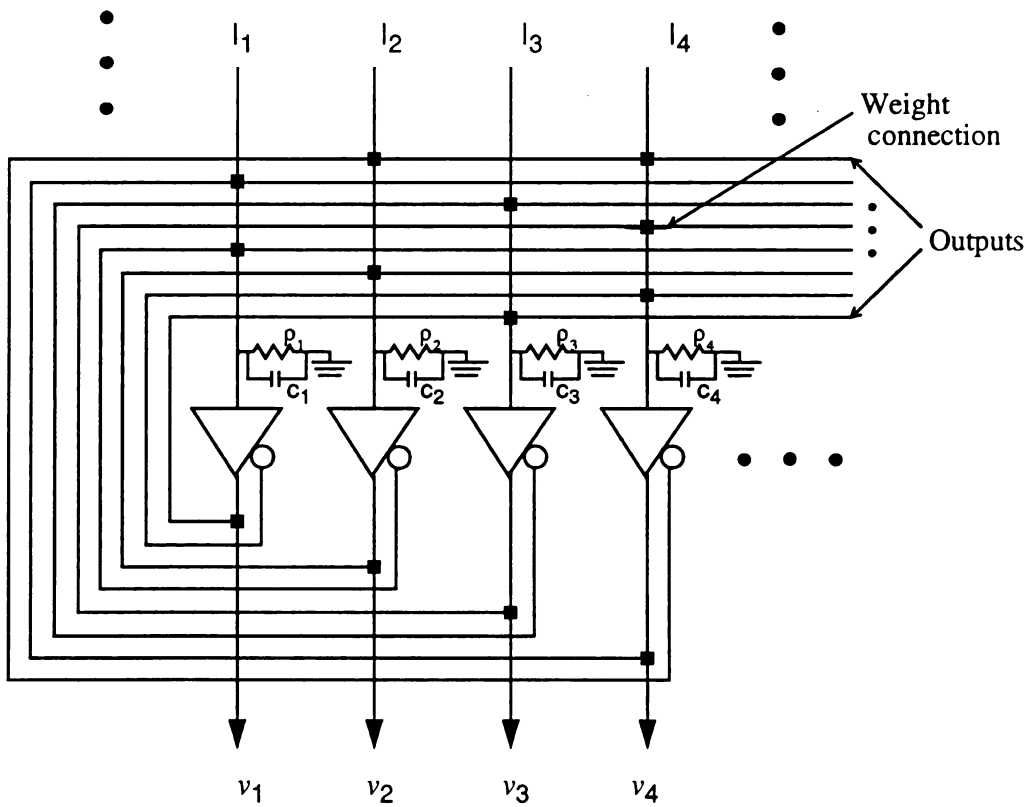


Figure 2.5. Hopfield network model.

The dynamics of an interacting system of n neurons can be described by the nonlinear differential equation

$$C_i \frac{du_i}{dt} = \sum_{j=1}^{n} w_{ij} v_j - \frac{u_i}{R_i} + I_i \tag{2.2}$$

where

$$\frac{1}{R_i} = \frac{1}{\rho_i} + \sum_{j=1}^{n} w_{ij},$$

I_i is an external input current, $v_j = f_j(u_j)$, and f_j is a sigmoid function. $R_i C_i$ forms the time constant of neuron i for charging and discharging and u_i/R_i is the leakage current. The energy function defined by integral of equation 2.2 is

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} v_i v_j - \sum_{i=1}^{n} I_i v_i + \sum_{i=1}^{n} \frac{1}{R_i} \int_0^{v_i} f^{-1}(\xi_i) d\xi_i \tag{2.3}$$

for $C_i \frac{du_i}{dt} = -\frac{\partial E}{\partial v_i}$.

If $w_{ij} = w_{ji}$ for all i and j, the time derivative of the energy function is

$$\frac{dE}{dt} = -\sum_{i=1}^{n} \frac{1}{C_i} \frac{df(u_i)}{du_i} \left(\frac{\partial E}{\partial v_i}\right)^2. \tag{2.4}$$

Since $f(u_i)$ is monotonically increasing, $\frac{dE}{dt} \leq 0$ for all t. As a result, the value of the energy function is strictly decreasing and becomes zero only at the equilibrium point where $\frac{dE}{dt} = -C_i \frac{du_i}{dt} = 0$ for all i.

Equations 2.2 and 2.3 define a gradient system and thus guarantee convergence. The Hopfield model has been applied to combinatorial optimization problems where it has been observed that the network model converges to a good solution in a few time constants [6, 10]. The objective function of the combinatorial problem is mapped to the computational energy function through the adjustments of the connectivity strengths w_{ij}. Local minima of the energy function correspond to solutions to the

problem. When the Hopfield network is used as an associative memory, solutions for this network model may be memory patterns stored in the network. Approximately $0.15n$ memory patterns are simultaneously stored before the patterns become too close to each other and tend to merge [4].

2.1.2.2 The Kennedy-Chua Model

A canonical circuit model with feedback was proposed for solving both linear and nonlinear programming problems by Kennedy and Chua [35, 36]. This model uses integrators as neuron elements. The structural parameters of the networks correspond to the coefficients of the objective function and constraints descriptions. Figure 2.6 shows an architecture of the model, where p-cells are constraint amplifiers, $\int$-cells are integrators, and V is the node voltages $v_1, v_2, \cdots, v_n$. The network dynamics can be described by

$$C_i \frac{dv_i}{dt} = -\frac{\partial f}{\partial v_i} - \sum_{j=1}^{m} p_j(g_j(V)) \frac{\partial g_j}{\partial v_i} \tag{2.5}$$

where C_i is capacitance, v_i is the voltage of node i, $f(v)$ is the objective function, and g(V) are constraints. The corresponding energy function is

$$E(V) = f(V) + \sum_{j=1}^{m} \int_0^{g_i(V)} p_j(\xi) d\xi. \tag{2.6}$$

Since $\frac{dE}{dt} \leq 0$ for all t, E(V) is a Lyapunov function ensuring the system convergence to a stable equilibrium point without oscillation [36]. This model requires more hardware to form the integrator than the Hopfield model does, but it is superior to the Hopfield model in solving linear programming problems for which the Kennedy-Chua model guarantees a stable equilibrium point while the Hopfield model does not [79].

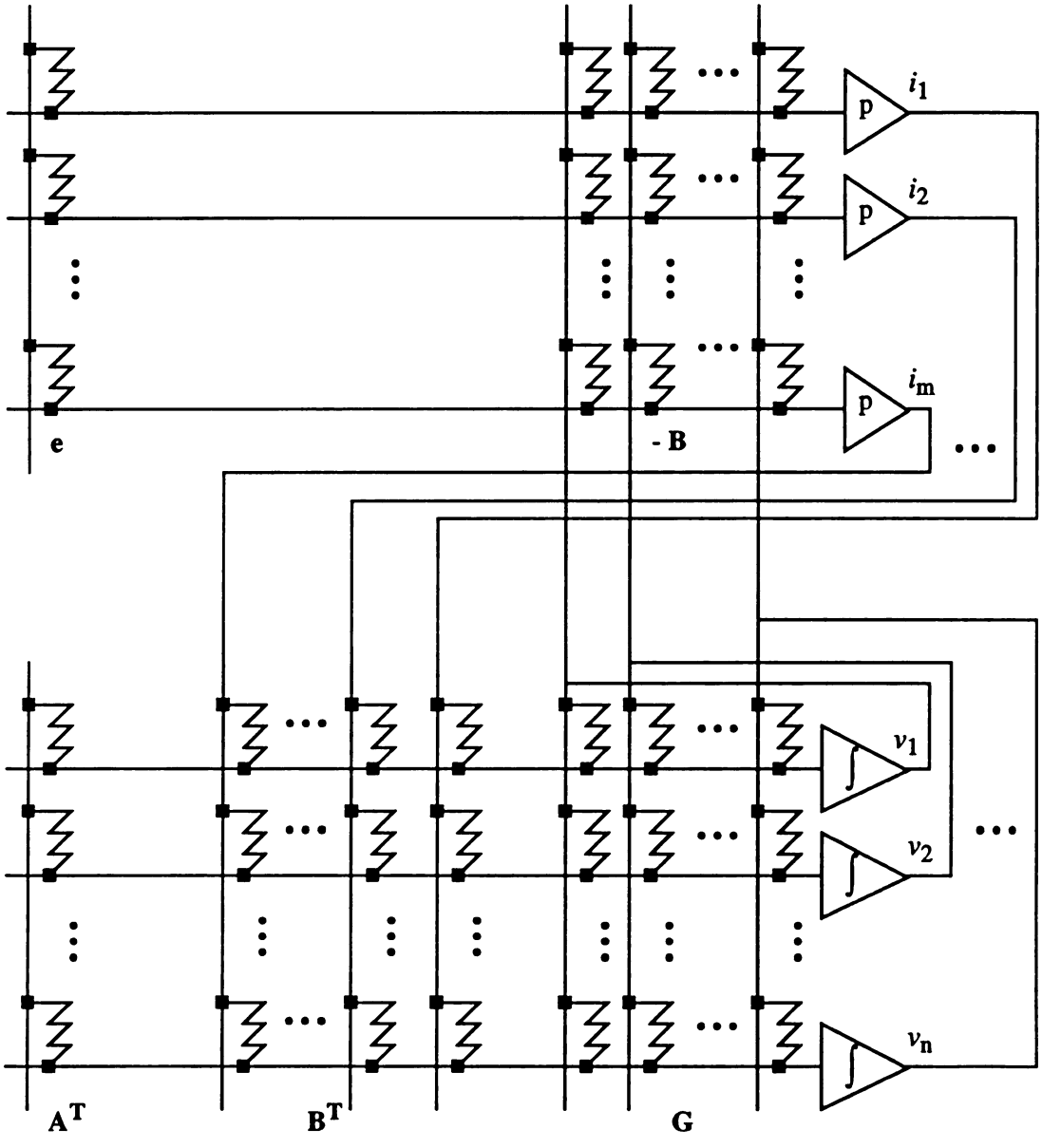


Figure 2.6. The Kennedy-Chua network model.

2.1.3 Feedforward Model

The Hopfield and Kennedy-Chua models are examples of one-layer feedback structures. The interconnection structures of biological neurons are often organized into multiple layers of cells [7, 33]. Layered feedforward networks were first studied in detail by Rosenblatt and his colleagues in the early 1960's [42]. Since then, feedforward multilayered structures and learning algorithms for training have been developed. The networks are trained with a set of input-target pairs as examples and can successfully generalize what has been learned. Feedforward networks have been applied to pattern recognition [37, 38, 49], robotics [39], and control problems [40, 41].

2.1.3.1 Simple Perceptrons

A simple perceptron is a single layered feedforward neural network, consisting of n inputs and an output layer. Figure 2.7 illustrates an example of a simple perceptron.

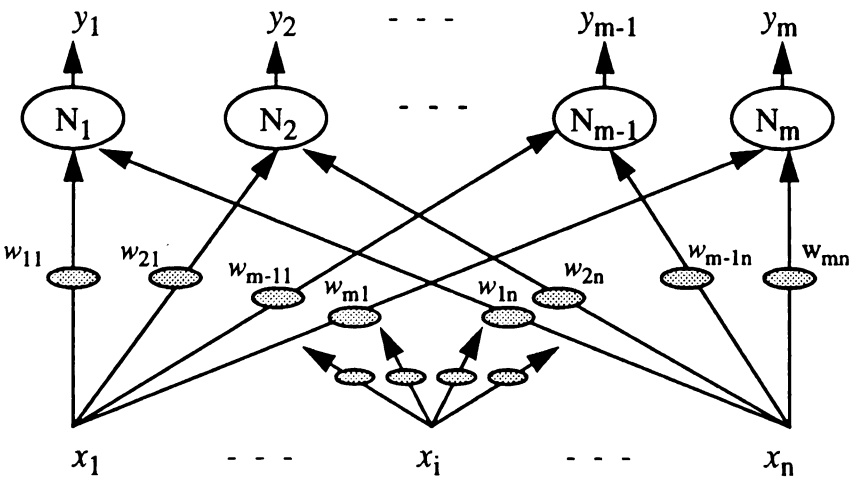


Figure 2.7. A simple perceptron.

x_i^μ is the ith element of the input pattern and y_i^μ is the output of neuron i when pattern μ is presented to the network. w_{ij} is the connection weight between neuron i and the jth element of the input pattern. If the number of patterns is p such that $\mu = 1, 2, \cdots, p$, the output in the output layer can be described by

$$\begin{aligned} y_i^\mu &= f(\sum_{j=1}^{n} w_{ij} x_j^\mu + \theta_i) \\ &= f(\sum_{j=0}^{n} w_{ij} x_j^\mu), \end{aligned}$$

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

When t_i^μ is the desired output of neuron i for input pattern μ, the cost function, which measures the system's performance, is defined by

$$\begin{aligned} E &= \frac{1}{2} \sum_{\mu=1}^{p} E^\mu \\ &= \frac{1}{2} \sum_i \sum_\mu (t_i^\mu - y_i^\mu)^2 \\ &= \frac{1}{2} \sum_i \sum_\mu [t_i^\mu - f(\sum_j w_{ij} x_j^\mu)]^2. \end{aligned} \tag{2.7}$$

The connection weights, w_{ij}, are changed by the gradient descent algorithm.

$$\begin{aligned} \Delta w_{ij} &= -\eta \frac{\partial E}{\partial w_{ij}} \\ &= \eta \sum_{\mu=1}^{p} (t_i^\mu - y_i^\mu) f'(\sum_j w_{ij} x_j^\mu). \end{aligned} \tag{2.8}$$

The condition for the existence of a solution in the simple perceptron is the linear independence of the input patterns [43]. The simple perceptron can not solve problems in which input patterns are not linearly independent, and may offer alternate partial solutions [43]. However, multilayer feedforward neural networks with nonlinear neuron elements can overcome this limitation.

2.1.3.2 Multilayer Perceptrons

A multilayer perceptron (or feedforward neural network) consists of an input layer, an output layer, and one or more hidden layers in between. Figure 2.8 shows the generic structure of a multilayer neural network. y_i is the output of neuron i and w_{ij} are connection strengths between neuron pairs. Outputs of any layer are weighted and summed as an input to a neuron in the next layer. An external input is applied to the input layer.

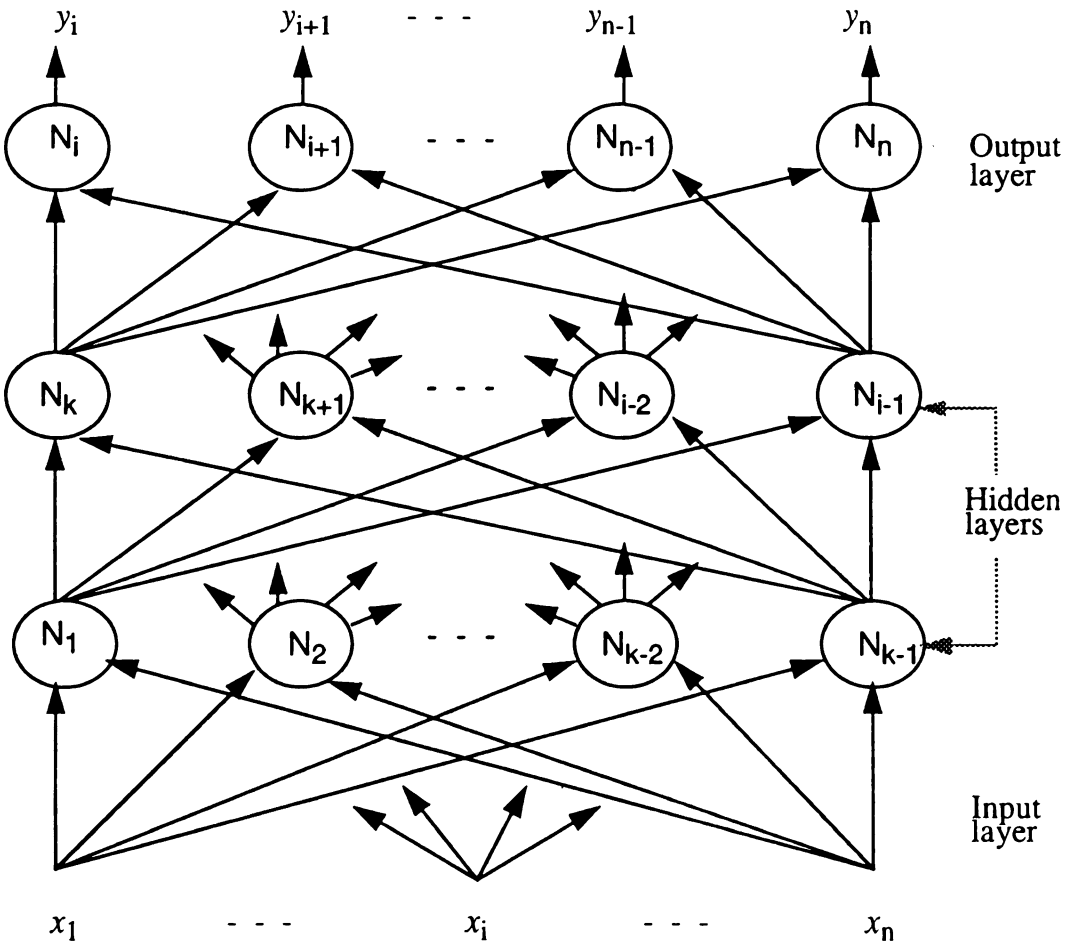


Figure 2.8. Multilayered feed-forward neural network.

Given pattern μ, where $\mu = 1, 2, \cdots, p$, a net input net_i^μ in neuron i in any layer is

$$net_i^\mu = \sum_j w_{ij} y_j^\mu \tag{2.9}$$

where y_j^μ is the output of neuron j in the previous layer when pattern μ is presented. $y_0^\mu = 1$ is often used. Thus, neuron i produces output

$$y_i^\mu = f(net_i^\mu) = f(\sum_j w_{ij} y_j^\mu) \tag{2.10}$$

where $f(\cdot)$ is a differentiable sigmoid function. For a given input pattern, the output of the output layer is compared to the target pattern and the connection weights between layers are modified in a backward direction according to the error. This is known as back-propagation learning. Given pattern μ, the error measure is

$$E^\mu = \frac{1}{2} \sum_i (t_i^\mu - y_i^\mu)^2 \tag{2.11}$$

where t_i^μ and y_i^μ are the desired output and actual output for the ith output neuron, respectively, when pattern μ is presented. The back-propagation rule states that

$$w_{ij}(k) \;=\; w_{ij}(k-1) \;+\; \sum_\mu \Delta_\mu w_{ij}(k) \tag{2.12}$$

For the output-to-hidden layer connections, the gradient descent rules gives

$$\begin{aligned} \Delta_\mu w_{ij}(k) &= -\eta \frac{\partial E^\mu}{\partial w_{ij}} \\ &= -\eta \frac{\partial E^\mu}{\partial net_i^\mu} \frac{\partial net_i^\mu}{\partial w_{ij}} \\ &= -\eta \frac{\partial E^\mu}{\partial y_i^\mu} \frac{\partial y_i^\mu}{\partial net_i^\mu} \frac{\partial net_i^\mu}{\partial w_{ij}} \\ &= \eta \delta_i^\mu y_j^\mu \end{aligned} \tag{2.13}$$

where $\delta_i^\mu = (t_i^\mu - y_i^\mu) f_i'(net_i^\mu)$.

In the hidden-to-hidden (or input) layer connections, $\Delta_\mu w_{ij}$ for the connection between neuron i in the hidden layer and neuron j in the lower layer can be obtained by using the chain rule.

$$\begin{aligned}
\Delta_\mu w_{ij}(k) &= -\eta \frac{\partial E^\mu}{\partial w_{ij}} \\
&= -\eta \frac{\partial E^\mu}{\partial y_i^\mu} \frac{\partial y_i^\mu}{\partial w_{ij}} \\
&= -\eta \sum_k \frac{\partial E^\mu}{\partial net_k^\mu} \frac{\partial net_k^\mu}{\partial y_i^\mu} \frac{\partial y_i^\mu}{\partial w_{ij}} \\
&= \eta \delta_i^\mu y_j^\mu
\end{aligned} \tag{2.14}$$

where $\delta_i^\mu = f_i'(net_i^\mu) \sum_k \delta_k^\mu w_{ki}$ and k denotes neurons in the upper layer.

The overall measure of the error is therefore

$$E = \sum_{\mu=1}^{p} E^\mu. \tag{2.15}$$

Thus, the back-propagation rule for any layer has the form

$$\begin{aligned}
\Delta w_{ij} &= -\eta \sum_{\mu=1}^{p} \frac{\partial E^\mu}{\partial w_{ij}} \\
&= \eta \sum_{\mu=1}^{p} \delta_i^\mu y_j^\mu.
\end{aligned} \tag{2.16}$$

Some variations of the ordinary back-propagation algorithm have been suggested in order to help the networks learn faster or escape local minima [45-47]. Multilayer feedforward networks trained by these back-propagation algorithms have been used to solve pattern classification problems [45, 48-50].

2.1.4 Recurrent Model

Recurrent networks allow connections in both directions between a pair of layers, and within a layer to itself. The Boltzmann machine is a well-known recurrent network with symmetric connections [51, 52].

2.1.4.1 Boltzmann Machine

The Boltzmann machine consists of visible and hidden units where the visible units can be divided into input and output units. Figure 2.9 illustrates the structure of the Boltzmann machine. The units are stochastic and take output value $v_i = +1$ with probability $f(h_i)$ and value $v_i = -1$ with probability $1 - f(h_i)$, where

$$h_i = \sum_j w_{ij} v_j$$

and

$$f(h) = \frac{1}{1 + e^{-2\beta h}}.$$

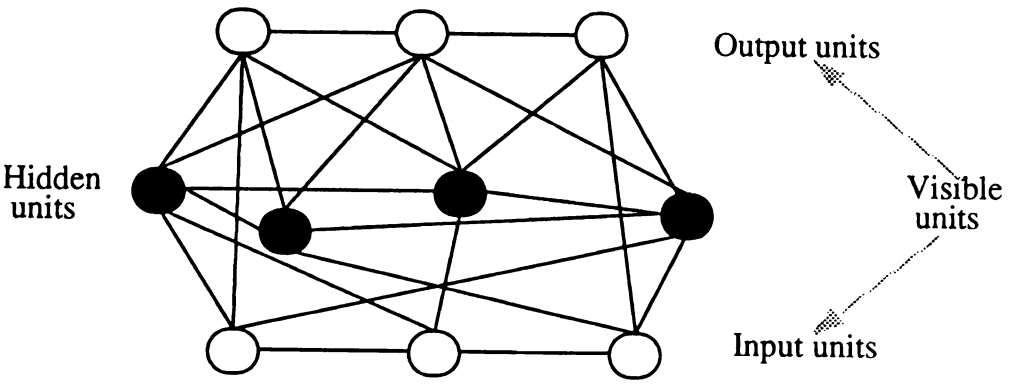


Figure 2.9. A Boltzmann machine consisting of visible and hidden units.

Here $\beta = \frac{1}{T}$ where T is pseudo-temperature. If $w_{ij} = w_{ji}$ for all i and j, the energy function

$$H\{v_i\} = -\frac{1}{2}\sum_i \sum_j w_{ij} v_i v_j \tag{2.17}$$

has a minimum at a stable state characterized by $v_i = sgn(h_i)$ where $sgn(h_i) = +1$ if $h_i \geq 0$, otherwise $sgn(h_i) = -1$.

The probability of finding the system in a particular state $\{v_i\}$, after equilibrium is reached, is given by the Boltzmann-Gibbs distribution

$$P\{v_i\} = e^{\frac{-\beta H\{v_i\}}{Z}}$$

where Z is a normalized constant.

Boltzmann learning adjusts the connections w_{ij} such that the states of the visible units, α, have a desired probability distribution. Let β be the states of the hidden units. The probability P_α of finding the visible units in state α irrespective of β is

$$\begin{aligned} P_\alpha &= \sum_\beta P_{\alpha\beta} \\ &= \sum_\beta e^{-\frac{\beta H_{\alpha\beta}}{Z}} \end{aligned} \tag{2.18}$$

where

$$H_{\alpha\beta} = -\frac{1}{2}\sum_i \sum_j w_{ij} v_i^{\alpha\beta} v_j^{\alpha\beta}.$$

The relative entrophy between actual probability P_α and desired probabilities R_α is

$$E = \sum_\alpha R_\alpha \log \frac{R_\alpha}{P_\alpha}. \tag{2.19}$$

$E \geq 0$ and $E = 0$ if $P_\alpha = R_\alpha$ for all α. The gradient descent rule gives

$$\begin{aligned} \Delta w_{ij} &= -\eta \frac{\partial E}{\partial w_{ij}} \\ &= \eta \sum_\alpha \frac{R_\alpha}{P_\alpha} \frac{\partial P_\alpha}{\partial w_{ij}} \\ &= \eta\beta[\sum_\alpha \sum_\beta R_\alpha P_{\beta|\alpha} v_i^{\alpha\beta} v_j^{\alpha\beta} - < S_i S_j >] \end{aligned} \tag{2.20}$$

where the correlations $< S_i S_j >$ are measured by taking a time average of $S_i S_j$ and the system must reach an equilibrium state for each α. A simulated annealing procedure is used to rapidly achieve a global minimum. Disadvantages of the Boltzmann machines are that learning requires an extremely long convergence time even with simulated annealing and its hardware implementation is impractical.

Boltzmann machines have been applied to various problems: statistical pattern recognition [7], constraint satisfaction problems [51], and combinational optimization [53].

2.2 Artificial Neural Network Implementations

Many current ANN models rely on software simulations run on serial or parallel digital computers. The speed of software simulation even on a parallel machine is far from equaling that of specialized hardware ANNs mainly because of programming and communication overhead. To date, a number of ANN hardware prototypes have been built using electronic, optical, and opto-electronic technologies. Electronic ANN hardware implementations, software simulators run on digital computers, and related issues are discussed. ANN implementations can be divided into three categories based on the method used to express the values within the network: analog, digital, and hybrid.

2.2.1 Analog and Hybrid Implementations

Analog computation performed in analog or hybrid electronic hardware uses some fundamental physical principles such as the linear attenuation of voltage by an electrical resistor and the nonlinear transfer characteristics of an amplifier. In a simple analog neural network, the interconnections are simple fixed value resistors (see Figure 2.5). The output voltage of neuron i is given by

$$v_i = f(\sum_{j=0}^{n} w_{ij} v_j)$$

where w_{ij} is the conductance of the resistor between neuron i and neuron j and $f(\cdot)$ is the transfer function of the amplifier. Neural networks with fixed value resistors can be used when the network function is known in advance and weight changes are not needed. This type of network with 256 neurons was designed on a single chip using standard CMOS technology by Jackel, *et al.* [15]. This circuit was not programmable due to the fixed synaptic weights.

ANNs can be programmed by storing synaptic weights in memory. A static memory cell has been used as storage for a weight bit where the neurons and synapses were binary units. Multiplication was performed by a logical XOR gate [16]. For many applications, a higher resolution for weight values is required. One way of storing analog weights is to use a capacitor [55, 56]. A weight can be stored as the voltage difference between two capacitors; the voltage difference is multiplied by the input voltage in the circuits. The main disadvantage of this dynamic storage technique is that it requires refresh circuitry to overcome the charge leakage on the capacitance.

An alternate way is to store weights digitally. In this case, a digital-to-analog (D/A) converter is required at each connection to perform an analog multiplication of the stored weight with the input signal. A matrix with 1024 multiplying D/A con-

verters was built using CMOS technology, where a weight was represented in four-bit magnitude plus a sign bit [57].

A floating gate field effect transistor (FET) was used as a device to combine the weight storage and the multiplication, where the weight was determined by the charge stored in the floating gate. However, the weight range and polarity difficulties were significant limitations [58]. To overcome these difficulties, a Gilbert multiplier [59] was used to carry out the weight multiplication while a floating gate FET was used simply for weight storage [60]. Sage and his associates designed an ANN chip based on Metal Nitride Oxide Semiconductor (MNOS) floating gate transistor technology and Charge Coupled Device (CCD) technology [14]. Analog weights were stored in MNOS floating gate transistors. Charge packages instead of currents were added to compute the sum of products. This circuit implemented a simple Hopfield-type neural network by operating with binary inputs and analog weights.

In analog computation, available mathematical functions are limited because those functions are found in some physical principles of devices. When a complex transfer function is required, it is difficult to implement correctly using analog hardware alone. In this case, hybrid ANN hardware is more appropriate where the sum of products is carried out with analog components, digitized for the transfer function processing, and then converted back to analog [24].

The potential advantage of analog computation is that operations in the network can be performed using inexpensive hardware. However, analog computation results in low accuracy and limited dynamic range due to physical constraints, such as thermal and quantum noise of analog components. In addition, design flexibility in analog implementation is strictly constrained because only mathematical functions resulting from physical principles are available for use.

2.2.2 Digital Implementations

In this section, the two mainstream approaches to digital implementation of ANNs are discussed: software simulations on general-purpose or special-purpose computers and dedicated VLSI implementation.

2.2.2.1 Software Simulators

ANN simulations on digital computers can be divided into two categories: ANN simulations on general-purpose parallel computers and ANN simulations on special-purpose processors.

Many general-purpose parallel machines, consisting of a large number of processing elements, are currently used for ANN simulation. Processing elements, cooperated on the same task, communicate through a single high speed data path between processing elements. A neural network and data are partitioned into different processing elements. Each processing element may have a dedicated memory to store data assigned. For example, the Warp machine, which was a systolic array of 10 processing elements, was used to implement a back-propagation network [61]. Each processing element contained an adder, a multiplier, and an ALU. The 39 Mbyte cluster memory was used to store weights and 17 million weight updates per second was achieved. Forrest, *et al.* used a Distributed Array Processor (DAP) consisting of 4096 processors to implement a Hopfield network [62]. The DAP was able to perform 25 million additions per second. The use of general-purpose parallel machines for ANN simulations can be justified for the problems to be completed in a feasible amount of simulation time. However, large-size ANNs often require faster simulations.

Special purpose processors, which are designed for ANN simulations and attached

as coprocessors to a host computer, are often called neurocomputers. A user program run on the host computer calls a special subroutine, and controls the neurocomputer whenever needed. Three methods for attaching a neurocomputer to a host computer have been defined [58]. The first method is to install the neurocomputer as a memory-mapped device on the host computer. In this method, the neurocomputer shares the memory space of the host computer. Data transfers between the host computer and the neurocomputer are controlled by the central processing unit in the host with addresses in the memory space. The second method is to attach the neurocomputer as a peripheral device using a standard peripheral interface . The neurocomputer can be ported from one type of host computer to another relatively easily. The first and second methods have high bus loading problems on the host computer. In addition, the second method suffers from the reduced bandwidth of the peripheral interface. Thus, these two approaches are appropriate for small computers. The third approach is to attach the neurocomputer as a coprocessor to a host computer via a local area network (LAN). This method has the advantage that the neurocomputer can access memory servers and other outboard devices on a high-bandwidth LAN.

Several manufacturers, such as TRW, Science Applications International Corporation, and Hecht-Nielsen Neurocomputers, developed neurocomputers. For example, Mark III and Mark IV neurocomputers were developed by TRW [63]. The Mark III (Mark IV) machine consisted of many Motorola 68010 (68020) based single board computers mounted on a broadcast bus backplane. These systems used the Artificial Neural System Environment (ANSE) developed at TRW for specifying the neural network to be implemented. A neural network was called on the Mark III (IV) from user software on the DEC Micro VAX through an user interface. The Mark IV had an ultra high-speed graphics display facility for monitoring the activity of the neural network. The Mark III and Mark IV systems were able to process up to 450,000 and 5,000,000 interconnections per second, respectively.

2.2.2.2 Dedicated Hardware Implementations

In order to fully utilize the parallelism embedded in ANN computations, the design of dedicated VLSI ANN digital systems is desired. A three-layer feedforward ANN was designed to classify handwritten numbers [20]. The network consisted of 50 neurons and 6688 fixed interconnections using a 2-micron CMOS process. The resulting VLSI layout was 7.9 x 9.2 mm^2 in size. This design is quite compact, but its flexibility was so low due to the fixed synaptic weights. Suzuki and Atlas mapped an ANN to an array of custom processors [64, 65]. Figure 2.10 shows the structure of the proposed processing element, where the blocks represent special operations for the network update. A weight matrix W and a threshold vector θ are stored in the product-sum unit (PSU). The arithmetic unit (AU) performs operations required for back-propagation.

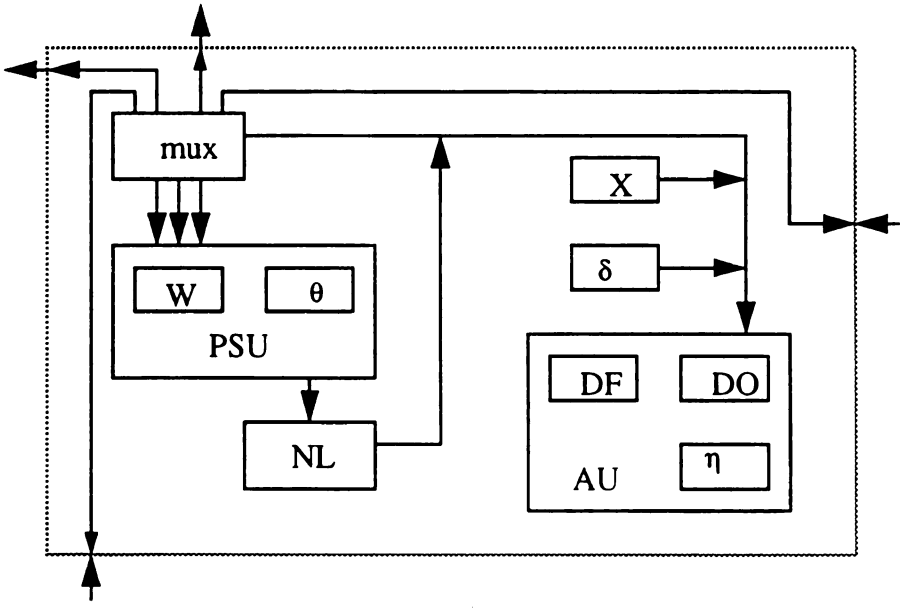


Figure 2.10. The structure of a processing element in [64].

The derivative of a nonlinear function (DF), desired outputs (DO), and a learning rate (η) are stored in the memory of the AU. Neural activations (X) and the error value (δ) are accessed by both the PSU and the AU. This ANN hardware has a high design flexibility, but hardware requirements for this design are large.

As indicated in the above two examples, dedicated digital ANN implementations can facilitate high parallelism, but it is difficult to simultaneously achieve the desired high design flexibility and high density on a chip.

A new digital approach - digital ANNs using stochastic computing techniques - replaces algebraic operations in ANNs by stochastic processes using pseudo-random pulse sequences [28, 31, 32]. Simple logic gates combined with other digital components perform multiplications and nonlinear transformation of signals.

In this new approach, the values for synaptic weights and input operands are normalized after a network has been trained [28, 32] or all operands are restricted to the range between 0.0 and 1.0 both for training and testing [66]. An operand x in the pulse-mode representation is the probability of pulse occurrence in the corresponding binary sequence $x_{(n)}$ at each clock. $\hat{x}$ is the estimate of x taken over finite clock periods N. Stochastic computations using random pulse sequences inherently utilize concurrent processing in all synaptic and neuron elements. Furthermore, the use of simple logic gates as computing elements allows a high neuron-density on a chip and a relatively compact network architecture. High design flexibility can also be achieved by making the network programmable. However, network speed depends on the length of a sampling clock period. The sampling clock period is the time required to estimate the computation results. A longer sampling clock period yields more accurate computations. Thus, there exists a trade-off between speed and accuracy in this approach. Details on the network architecture, analysis, and performance will be discussed in following chapters.

2.3 Pattern Recognition and Neural Networks

Pattern recognition is concerned with classification or description of complex patterns by means of some measured properties. A pattern recognition system requires data acquisition, data representation, and data classification.

The design of a pattern recognition system involves the following three steps: (1) data acquisition, (2) preprocessing, and (3) decision making [68]. A typical character recognition system is illustrated in Figure 2.11.

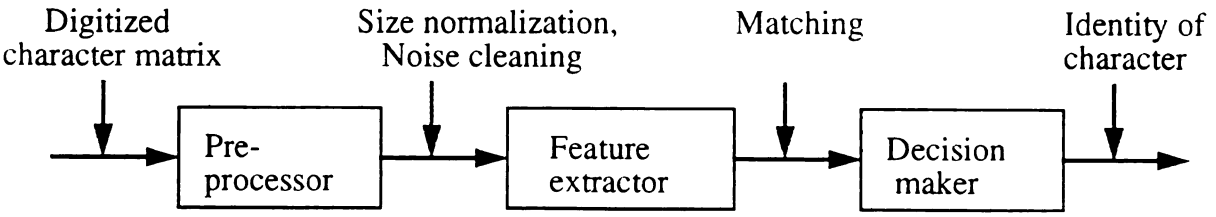


Figure 2.11. A typical character recognition system.

The first stage involves image processing, the last two stages deal with the pattern recognition. Mask (joint occurrences of black and white pixels), strokes and bays in various directions, the location of end points, and the intersection of line segments and loops, are all popular features for character recognition. Most pattern recognition systems utilize one of the following three approaches: statistical, structural, or neural network.

2.3.1 Statistical Approach

In the statistical approach, a pattern is represented in terms of N features. Each

pattern can be viewed as a point in the N-dimensional space. If the choice of features is good, then pattern vectors belonging to different classes will occupy different regions of this feature space. The objective in this approach is to establish decision boundaries in the feature space to separate patterns belonging to different classes.

Assume that a given sample pattern belongs to one of M classes $c_1, c_2, \cdots, c_M$ based on its feature vector $\mathbf{x} = (x_1, x_2, \cdots, x_N)$ and that $\mathbf{x}$ has a class-conditional density $p(\mathbf{x}|c_i)$. Bayes decision rule states that a pattern with $\mathbf{x}$ as its feature vector is assigned to class c_i if

$$p(c_i|\mathbf{x}) \geq p(c_j|\mathbf{x}) \quad for \ all \ j \neq i$$

where $p(c_i|\mathbf{x})$ is the *posteriori* density for class c_i, defined as

$$p(c_i|\mathbf{x}) = \frac{p(\mathbf{x}|c_i)p(c_i)}{\sum_{i=1}^{M} p(\mathbf{x}|c_i)p(c_i)}$$

where $p(c_i)$ is *a priori* probability density for class c_i. If $p(c_i) = 1/M$, then the Bayes decision rule is identical to the maximum-likelihood decision rule. The decision boundary between pattern class c_i and c_j is defined by

$$p(c_i|\mathbf{x}) - p(c_j|\mathbf{x}) = 0.$$

If class-conditional densities are multivariate Gaussian, then

$$p(\mathbf{x}|c_i) = N(\mu_i, I),$$

where μ_i is the mean vector for class c_i and I denotes the identity covariance matrix.

If $p(c_i)$ for all i are equal, then

$$p(c_i|\mathbf{x}) = -\frac{||\mathbf{x} - \mu_i||^2}{2},$$

where $||\cdot||$ denotes the Euclidean norm. As a result, a pattern $\mathbf{x}$ is assigned to the class of the closest mean vector. If the class-conditional densities are known, Bayes decision rule can be used to design a classifier. If they are not known, they must be estimated by training with sample patterns.

2.3.2 Structural Approach

When the number of features required to establish a reasonable decision boundary is very large, it is more appropriate to view a pattern as being composed of simple subpatterns. In the structural approach, a complex pattern is represented in terms of the interrelationships among the simplest subpatterns, called primitives. This paradigm has been used in situations where the patterns have a definite structure which can be captured in terms of a set of rules.

The primitives or grammatical rules must be inferred from the available samples. In this approach, the difficulty resides in segmentation or reliable extraction of the primitives from a finite number of pattern samples.

2.3.3 Neural Network Approach

The neural network approach is based on the notion that a network of simple processing elements arranged in a manner similar to a biological neural system might

be able to self-organize itself to recognize and classify patterns. The Perceptron is considered as the first significant development of such in the early 1960s [2]. The basis for the inherent power of Perceptron devices was well understood. However, at that time, no method was known for training multilayer Perceptron devices and the cost for full implementation of those devices was extremely high. VLSI technology has advanced and the price of processors has dropped tremendously. More significantly, the generalized delta rule developed in 1986 by Rumelhart, *et al.* provides a practical way for training the multilayer Perceptrons [8]. Today, perceptron-like models trained by the generalized delta rule are being applied to pattern recognition.

In pattern recognition systems using the neural network approach, all stages or some of stages in Figure 2.11 can be combined into one neural network. The network learns the mapping from the observation space to the interpretation space by a training algorithm. In this approach, human interactions involved in statistical or structural pattern recognition systems are minimized. Most recognition processes are performed in an autonomous manner.

2.4 Behavioral Modeling with VHDL

In the design of large systems like ANNs, use of Design Automation (DA) becomes necessary. The simulation and verification of a design using a behavioral description language at an early stage of the design process also becomes more important as the complexity of systems continues to grow. VHDL is a typical behavioral description language which is semantically oriented for digital systems. Digital ANNs can be modeled and simulated using VHDL.

2.4.1 Behavioral Modeling

A promising approach for implementing artificial neural networks is the fabrication of special-purpose VLSI chips. Traditionally designers start with a gate-level or a circuit-level schematic. However, as systems become more complex, a top-down design approach is needed in order to manage complexity and to reduce the design time and development costs. Test and modification of an original design can be done in an early stage of the design process. Top-down design starts with a high-level specification which is decomposed into lower level specifications in a hierarchical fashion. Designers look at the system at an abstract level in a high-level specification. Hardware Description Languages (HDLs) are crucial to the high-level design [69-72].

VHDL is a typical HDL that can be used to express the function and logical organization of circuits, ranging from simple logic gates to complex digital systems [73-77]. VHDL is fast becoming an industry standard. The U.S. government made it a standard language, requiring the use of VHDL as the design and description mechanism in Department of Defense (DoD) hardware designs. Compilers, translating the structural design in VHDL to an intermediate format such as Caltech Intermediate Format (CIF), are being produced by many CAD vendors.

In VHDL one can 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. For example, when an ANN is modeled in VHDL, its behavior may be described by a set of static or dynamic equations by using function statements. Structure is described by interconnections of the subcomponents (synapses and neurons). An efficient and precise modeling of VLSI ANNs is facilitated by analysis of VHDL semantics, including a detailed investigation of process statements, functions, and delay characteristics.

2.4.2 VHDL Characteristics

The primary element in VHDL is a design entity which can represent portions of a hardware design ranging from simple logic gates to complex digital systems. A design entity consists of two different types of descriptions: the entity declaration and one or more architectural bodies. The entity declaration defines the interface between the entity and the outside world. Figure 2.12 illustrates an example entity declaration.

```
entity COUNTER is
        generic (time_delay: time:= 10 ns);
        port (clk, reset: in bit;
              sum: buffer integer);
end COUNTER
```

Figure 2.12. Entity declaration in VHDL.

The ports are the signals through which the design entity communicates with other modules. Their declaration can be any predefined or user-defined type. The port and local item defined in the entity declaration are made available to architectural bodies associated with this entity. A set of parameters, called generics, provides a channel for static information to be communicated to a design entity from its environment. Generics can be used to specify timing characteristics, the bit size of ports, or other descriptive characteristics of a design such as temperature, capacitance, location, *etc.*

An architecture body supports three implementation styles of a design entity: behavioral, structural, and data-flow. The behavioral body describes the system model in sequential program statements just like programs written in a high-level program-

ming language. The structural body describes a design entity purely in terms of its subcomponents and their interconnections. Finally, the data-flow body decomposes the architecture into a set of concurrent register assignments under the control of gating signals. Data-flow style emphasizes the flow of information between memory and gating elements. All three styles may be intermixed in an architectural body.

A VHDL design entity is a template to be used in creating specific instances of a component via the component instantiation statement. A component may represent a structural partitioning of the design or a functional decomposition of a large system. Because this feature essentially isolates one level of design from another, two different design methodologies can be accommodated: top-down approach and bottom-up approach. In the former approach, the architectural body can be written in terms of abstract lower-level components. Such components must be fully described with a variety of design entities later in the code. In the latter approach, the local component declaration specifies the portion of the interface from an existing design entity that resides in the design library.

Designers may specify the behavior of a subsystem and leave the implementation details of structural design to others. Thus, VHDL designers can model simply the function of the system independent of any implementation technology.

A VHDL description is evaluated when an event occurs at one of the component's inputs. The evaluation yields a new set of projected values for the outputs of the component. This effect may, in turn, causes additional changes. Independent sequences of events can occur simultaneously. The event-driven semantics of VHDL are based on the assumptions that all signals in a design propagate in well-defined directions and that signal propagation always includes a delay.

A typical signal assignment statement consists of a driver and a target. A driver is a source of the value for a signal. A signal may have multiple sources. If a signal has more than one source, then all sources can participate in the calculation of the

value. Such a signal must be a resolved signal, and the resolution function calculates one effective value from an array of values. The target of a signal assignment is the signal on the left hand side of the assignment operator. The simulator creates a driver for each element of a target of every concurrent signal assignment [74].

Timing is one of the most important aspects of a VHDL model. The representation of time in VHDL has both a macrotime scale and a microtime scale. The macrotime scale represents real time (nanoseconds, microseconds, *etc.*) which is measured in discrete units. The microtime scale represents a unit delay which is essentially not measurable. Any number of micro-units of time may exist between any two macro-units of time. With two time scales, designers can perform unit-delay or real-time simulations [73].

There are two kinds of statements in VHDL: sequential and concurrent. Sequential statements are used to define algorithms for the execution of a subprogram or process. They are executed one at a time. Concurrent statements are executed in an asynchronous pseudo-parallel fashion. They are used to define interconnected blocks and processes that jointly describe the overall behavior or structure of a design.

Figure 2.13 shows the flow of data in the design process under a VHDL hardware support environment including an analyzer, a profiler, and a simulator. The design library contains intermediate representations of VHDL descriptions. The library unit resulting from the analysis of a design unit is placed into a working library. Only one library may be the working library during the analysis of any given design unit [74].

The analyzer accepts a VHDL source code, translates it into the intermediate form, and stores it in the design library. It checks the syntax and semantic rules of the language. The profiler pulls all necessary design entity interfaces, bodies, functions, and packages from the library, then configures a cross-section of a design hierarchy. The simulator and other tools may use this configuration. The simulator records signal histories and dynamic errors.

An understanding of VHDL semantics and characteristics enables designers to use VHDL as an economical hardware design testbench. A system can be first modeled behaviorally with a high-level specification using appropriate modeling techniques, verifying the correctness of the design. Later, the high-level specification can be decomposed into lower level specifications, incorporating more implementing technological constraints. Finally, when the system is modeled in complete structural descriptions, the precise feasibility and detail of a hardware realization can be assessed.

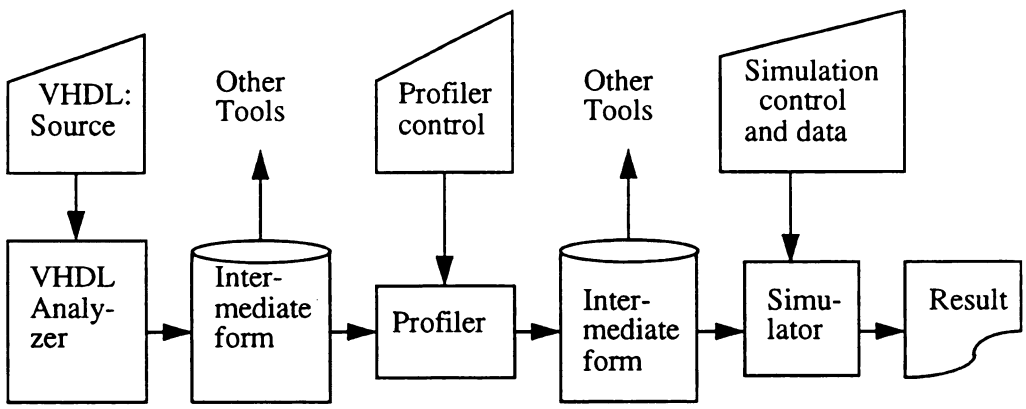


Figure 2.13. The flow of design data in VHDL design process.

CHAPTER 3

Stochastic Computing in Neural Networks

An approach to performing arithmetic operations using random pulse sequences is discussed. In this approach, a number is normalized into a fraction from 0 to 1. The fractional number is encoded using a random pulse stream where it is represented by the probability of a pulse occurrence in each clock period. Algebraic operations are replaced by stochastic processes, and computational results expressed as probabilities are estimated in finite clock periods. Inaccuracies are inherently associated with stochastic computing and can be described in terms of mean and variance. In this chapter, the method for generating random pulse streams is discussed and a new statistical model for the estimate of probability generated from a random pulse generator is developed. Stochastic computing techniques, which can be utilized in digital artificial neural networks, are presented.

3.1 Introduction

Von Neuman first observed that normalized numbers or voltages could be represented by probabilities and that some properties of the nervous system could be explained through statistics [80]. He intended to show that simple algebraic operations such as addition and multiplication could be performed by simple logical gates. Later, stochastic computing techniques using random pulse streams were proposed in the 1960's [81, 82].

In stochastic computation, 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. At the time it was originally proposed in the 1960's, integration technology was not mature and the hardware cost for arithmetic devices was expensive. A main objective in using stochastic computing techniques was to implement some algebraic computations by inexpensive large parallel processors at the cost of speed and accuracy. Since then, the hardware cost of digital computing elements has continued to drop as VLSI technology has advanced tremendously. Consequently, the idea of stochastic computing had been discarded.

However, the idea has been resurgent as an alternative to deterministic computations in the area of artificial neural networks since late 1980's. The main reason is that stochastic computing using random pulse sequences shares one very important characteristic with ANN dynamics: network performance depends not on the accuracy of calculations performed in an individual processing element, but on the collective properties of the network (or system) where each processing element does not nec-

essarily perform correct computations. Recently, some neural network architectures have been proposed based on this idea and applied to some engineering problems such as associative memory [28] or binary classification [32].

3.2 Generating Probability

3.2.1 Pseudo-Random Pulse Sequences

A pseudo-random pulse (or binary) sequence can be generated by a tapped Linear Feedback Shift Register (LFSR) [67]. Figure 3.1 shows the diagram of an n-bit LFSR.

The feedback function $f(x_1, x_2, \cdots, x_n)$ is expressed in the form

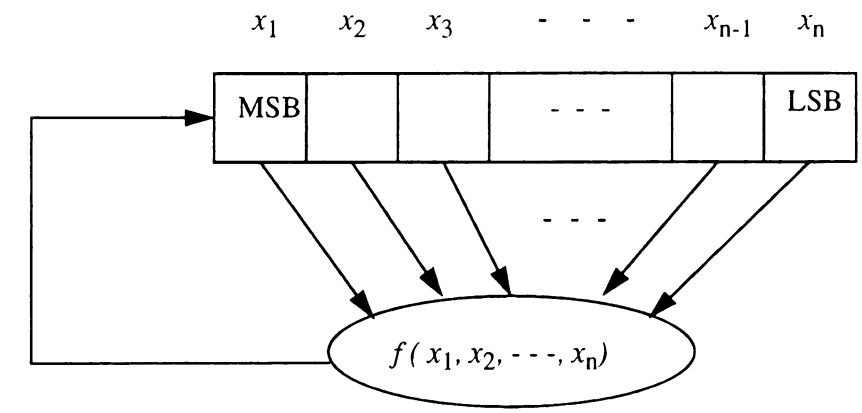
$$f(x_1, x_2, \cdots, x_n) = c_1 x_1 \oplus c_2 x_2 \oplus \cdots \oplus c_n x_n$$

where each constant c_i is either 1 or 0, the symbol $\oplus$ denotes modulo-2 adddition, and x_1 and x_n indicate the values of the most significant and least significant bits, respectively. For a given register length n, the maximal length period of a sequence is $p_{max} = 2^n - 1$.

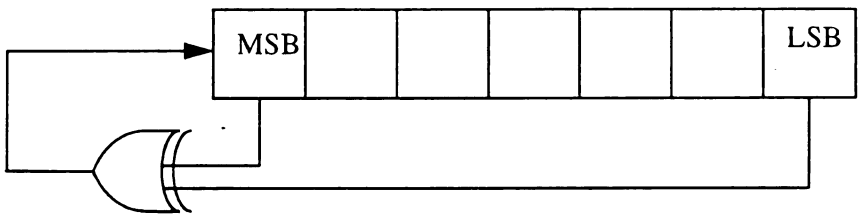
Define $\{a_n\}$ be a PN sequence if and only if it is a binary sequence satisfying a linear recurrence

$$a_k = \sum_{i=1}^{n} c_i a_{k-i} \ \ (modulo\ 2)$$

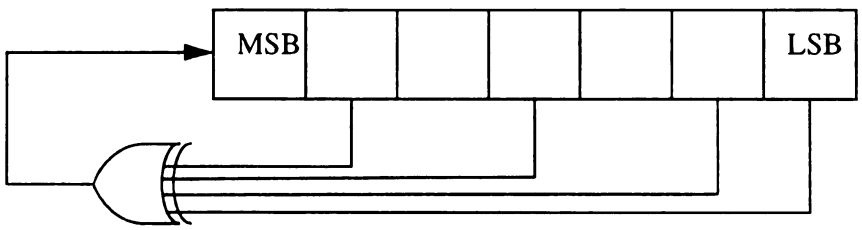
and has p_{max} as a period. There are 2^n combinations to select c_i's. Only a limited number of c_i combinations can form the maximal length PN sequences. In order to form a maximal length PN sequence, c_i is determined by the primitive polynomial [67].



(a)



(b)



(c)

Figure 3.1. (a) The block diagram of Linear Feedback Shift Register (LFSR); examples of LFSRs with a maximal length period where (b) $\{c_1, c_2, \ldots, c_7\} = \{1000001\}$ and (c) $\{c1, c_2, \ldots, c_7\} = \{0101011\}$.

Table 3.1. Number of distinct PN sequences with the maximal length period.

n	p_{max}	N_{pn}	n	p_{max}	N_{pn}
1	1	1	8	255	16
2	3	1	9	511	48
3	7	2	10	1023	60
4	15	2	11	2047	176
5	31	6	12	4095	144
6	63	6	13	8191	630
7	127	18	14	16383	756

Table 3.1 shows the number of distinct PN sequences, N_{pn}, with period $p = p_{max}$ with respect to the LFSR register length. For a given register length n, a sequence $\{a_n\}$ has the following random properties assuming that each 0 and 1 is replaced by 1 and -1, respectively [67]:

1. The number of 1's is nearly equal to the number of -1's in a maximal period p_{max}. More precisely,
$$|\sum_{n=1}^{p_{max}} a_n| \leq 1.$$

2. Every possible array of n consecutive terms occurs exactly once, except all 0's. This indicates that all n-bit integer numbers from 1 to $2^n - 1$ are generated exactly once in p_{max}.

3. The autocorrelation of a_n is
$$C(\tau) = \frac{1}{p_{max}} \sum_{n=1}^{p_{max}} a_n a_{n+\tau} = \begin{cases} 1 & \text{if } \tau = 0 \\ -1/p_{max} & \text{if } 0 < \tau < p_{max}. \end{cases}$$

These random properties of a_n are utilized for encoding fractional numbers into corresponding random pulse streams.

3.2.2 Generating Probability

In order to utilize stochastic computing techniques, the values of all operands must lie between 0 and 1. The fractional numbers represented by probabilities are encoded in random pulse streams. If the fractional number is stored in an n-bit register, the resolution is $\frac{1}{2^n-1}$. For example, when $n = 8$, 0.0 is stored as '00000000', 1/255 as '00000001', 2/255 as '00000010', *etc.* The random pulse stream corresponding to a fractional number can be generated by comparing the number with a pseudo random number. The pseudo random numbers can be generated from a PN sequence by taking all bits of the LFSR in parallel, as indicated in property 2 of the PN sequence. Fractional numbers from $\frac{1}{2^n-1}$ to 1 equally spaced by $\frac{1}{2^n-1}$ 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.

Generating probability x is defined as the probability of pulse occurrence in the corresponding random pulse sequence $x_{(n)}$ at each clock. x is estimated in a sampling clock period, where the sampling clock period is defined as finite clock periods taken for estimation of x. Error (or noise) is involved in estimating the generating probability in finite clock periods. Thus, estimate $\hat{x}$ can be modeled as an original signal plus random noise.

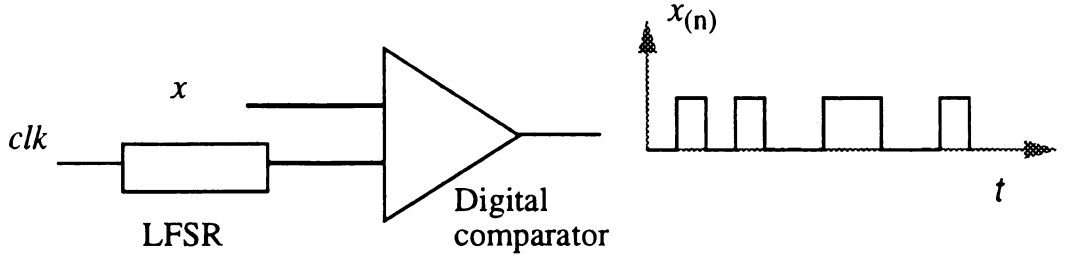


Figure 3.2. A random pulse generator for fractional number x.

3.3 Distribution of Estimated Generating Probability

The generating probability of the random pulse generator has been modeled as a binomial distribution in the literature [82, 83]. However, the distribution can be more precisely modeled by regarding the estimate as a hypergeometric random variable. This new model of the estimate is important, especially for case when a short sampling clock period is taken for estimation.

3.3.1 Factorial Moment Generating Function

The following terms are defined:

x: A fractional number or a generating probability.

$x_{(n)}$: The pseudo-random pulse sequence for x.

N: The period of $x_{(n)}$ such that $N = 2^n - 1$, where n is the order of a maximum length LFSR.

P_s: The sampling clock period for estimation.

X: A capital x is a discrete random variable indicating the number of logic level '1' pulses occurring in $x_{(n)}$, where $0 \leq X \leq P_s$ such that $X = 0, 1, 2, \ldots, P_s$.

$E(X)$: The expected value of X.

$Var(X)$: The variance of X.

$\hat{x}$: The estimate of x and a random variable such that $\hat{x} = X/P_s$.

The factorial moment generating function of the distribution of a random variable X is formally defined as follows [84]:

$$\eta(t) = E(t^X) = \int_{-\infty}^{\infty} t^x f_X(x) dx. \tag{3.1}$$

Differentiating $\eta(t)$ k times and substituting 1 for t gives

$$\begin{aligned} \eta^{(k)}(1) &= \frac{d^k}{dt^k} E(t^X)|_{t=1} \\ &= E[X(X-1)\cdots(X-k+1)], \end{aligned} \tag{3.2}$$

where $E[X(X-1)\cdots(X-k+1)]$ are called the *factorial moments.*

The variance of X can be computed from the first two factorial mements as

$$Var(X) = E[X(X-1)] + E(X) - [E(X)]^2. \tag{3.3}$$

3.3.2 Binomial Distribution Model

If the occurrence of successive pulses in a sequence $x_{(n)}$ is statistically independent, the sequence is called a Bernoulli sequence. The random variable X of interest is the number of logic level 1's occurring in a sampling clock period P_s. X is a Bernoulli variable. Consider $X = k$ indicating k logic level 1's occur in P_s clock periods.

Let the probability of pulse occurrence at each clock period in $x_{(n)}$ be $x = p$. The probability of one particular sequence with k logic level 1's in n clock periods is

$$p^k(1-p)^{n-k}.$$

The number of sequences with k logic level 1's in n clock periods is the same as the number of ways of taking k objects at a time from n objects. The number is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

The quantity $\binom{n}{k}$ is called the *binomial coefficient.*

Thus, the probability function of X can be expressed by

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, \ldots, n. \tag{3.4}$$

The density function of X is

$$f_X(x) = \sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k} \delta(x-k).$$

The distribution of X is *binomial.*

The factorial moment generating function of a binomial distribution is obtained using the binomial theorem as follows:

$$\begin{aligned} \eta(t) &= E(t^X) \\ &= \sum_{k=0}^{n} t^k \binom{n}{k} p^k (1-p)^{n-k} \\ &= \sum_{k=0}^{n} (pt)^k (1-p)^{n-k} \\ &= [pt + (1-p)]^n. \end{aligned} \tag{3.5}$$

The mean and variance of X can be computed using the first two factorial moments, $\eta'(1)$ and $\eta''(1)$ as

$$E(X) = np \quad \text{and} \quad Var(X) = np(1-p). \tag{3.6}$$

Thus, the mean and variance of $\hat{x}$ are, respectively

$$E(\hat{x}) = p \quad \text{and} \quad Var(\hat{x}) = \frac{p(1-p)}{n}. \tag{3.7}$$

3.3.3 New Distribution Model

A pseudo-random pulse sequence, $x_{(n)}$, has been modeled in the literature as a Bernoulli sequence [82, 83]. However, the pulse occurrence in $x_{(n)}$ is not perfectly independent because a maximum length LFSR generates fractional numbers between $\frac{1}{2^{-1}}$ and 1 such that each number occurs exactly once in a period. Accordingly, the pulse occurrence in $x_{(n)}$ has statistical dependency.

If $P_s = n$ and $x = p = l/N$, the probability that k '1' pulses in $x_{(n)}$ occur during the sampling clock period n is the same as the probability that k black balls are taken out in n withdrawals from the box containing l black balls and $N - l$ white balls, one ball being withdrawn at a time without replacement. Thus, the sampling distribution of X can be more closely modeled by the hypergeometric distribution. When $x = p = l/N$ and $P_s = n$, the probability function of X is

$$P(X = k) = \frac{\binom{l}{k}\binom{N-l}{n-k}}{\binom{N}{n}}$$

where l is a natural integer, *i.e.*, $l \in \{0, 1, 2, \cdots, N-1, N\}$.

Let $(k)_r$ be the product of r consecutive integers starting with k. Then, the rth factorial moment is

$$\begin{aligned} E[(X)_r] &= \sum_{k=r}^{N} (k)_r P(X = k) \\ &= \sum_{k=r}^{N} (k)_r \frac{\binom{l}{k}\binom{N-l}{n-k}}{\binom{N}{n}} \end{aligned}$$

$$= \frac{(l)_r(n)_r}{(N)_r}. \tag{3.8}$$

The detailed derivation of equation 3.8 can be found in Appendix A.

The expected value of X is

$$\begin{aligned} E(X) &= E[(X)_r]|_{r=1} \\ &= \frac{ln}{N} \end{aligned} \tag{3.9}$$

and for $r = 2$ in equation 3.8,

$$E[X(X-1)] = \frac{l(l-1)n(n-1)}{N(N-1)}. \tag{3.10}$$

From equations 3.9 and 3.10, the variance of X can be computed as

$$\begin{aligned} Var(X) &= E(X^2) - [E(X)]^2 \\ &= E[X(X-1)] + E(X) - [E(X)]^2 \\ &= \frac{l(l-1)n(n-1)}{N(N-1)} + \frac{ln}{N} - (\frac{ln}{N})^2 \\ &= n\frac{l}{N}\frac{N-l}{N}\frac{N-n}{N-1} \\ &= np(1-p)\frac{N-n}{N-1}. \end{aligned} \tag{3.11}$$

Thus, the expected value and variance of $\hat{x}$ are, respectively,

$$\begin{aligned} E(\hat{x}) &= \frac{E(X)}{n} \\ &= \frac{1}{n} \cdot \frac{nl}{N} \\ &= p \end{aligned} \tag{3.12}$$

and

$$\begin{aligned} Var(\hat{x}) &= \frac{1}{n^2} Var(X) \\ &= \frac{p(1-p)}{n} \frac{N-n}{N-1} \end{aligned} \tag{3.13}$$

where $0 \leq \frac{N-n}{N-1} \leq 1$ when $1 \leq n \leq N$. As noted from equation 3.13, when N is large (implying a wide LFSR) the distribution of $\hat{x}$ tends to the binomial distribution. If $P_s = N = n$, $Var(\hat{x}) = 0$ and $\hat{x}$ becomes a constant x.

This new statistical model is used to perform an analysis on random noise effects in digital multilayer neural networks (DMNN). The DMNN architecture will be developed in Chapter 4 and the analysis will be performed in Chapter 5.

3.4 Stochastic Computing in ANNs

Stochastic computing exploits the similarities between probability algebra and Boolean algebra. Logical operations with simple logical gates over multiple pulse sequences correspond to pseudo-analog computations.

3.4.1 Basic Stochastic Computations

A random pulse sequence $x_{(n)}$ is a sequence of pulses whose probability x can not be measured at any one clock period, but it can be approximated by a measurement of average pulse rate. Any Boolean operation over individual pulses corresponds to an algebraic operation among variables represented by their respective average pulse rates [82]. Figure 3.3 shows the duality between logical operations with actual pulse occurrences and numerical operations with pulse occurrence probabilities.

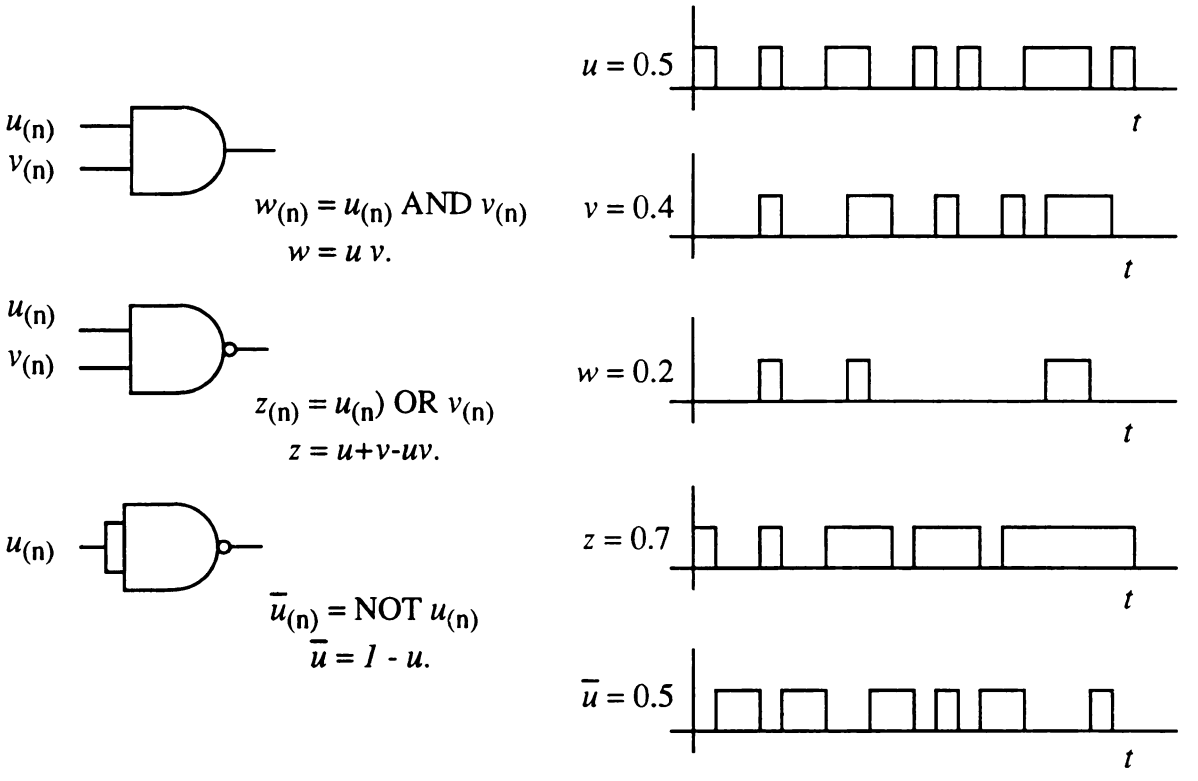


Figure 3.3. Dualtity between Boolean operations and numerical operations, where the sampling clock period = 20 is assumed and the number of '1' pulses generated during the period in $x_{(n)}$ is 20 x for x.

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

$$\begin{aligned} z &= P(z_{(n)} = 1) \\ &= P(x_{(n)} = 1 \ \wedge \ y_{(n)} = 1) \\ &= P(x_{(n)} = 1)\, P(y_{(n)} = 1) \\ &= xy \end{aligned} \tag{3.14}$$

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

$$
\begin{aligned}
z &= P(z_{(n)} = 1) \\
&= P(x_{(n)} = 1 \ \vee \ y_{(n)} = 1) \\
&= P(x_{(n)} = 1) + P(y_{(n)} = 1) - P(x_{(n)} = 1 \ \wedge \ y_{(n)} = 1) \\
&= x + y - xy. \qquad (3.15)
\end{aligned}
$$

Instead of being statistically independent, if two sequences are mutually exclusive, implying that no two pulses coincide in two random pulse sequences, $P(x_{(n)} = 1 \ \wedge \ y_{(n)} = 1) = xy = 0$ in equation 3.15. Thus the logical OR performs a direct summation.

The NOT gate in Figure 3.3 (c) produces an output pulse whenever no input pulse occurs. If $x_{(n)}$ is an input pulse sequence of a NOT gate, the probability of pulse occurrence in an output sequence $z_{(n)}$ is

$$
\begin{aligned}
z &= P(z_{(n)} = 1) \\
&= 1 - P(x_{(n)} = 1) \\
&= 1 - x. \qquad (3.16)
\end{aligned}
$$

A complete set of examples of stochastic computations utilizing the duality between Boolean operations and algebraic operations can be found in reference [82].

3.4.2 Stochastic Computing in the DMNN

Neural operations in a stochastic neural network of the type considered here 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

$$\begin{aligned} m_{ij} &= P(m_{ij(n)} = 1) \\ &= P(w_{ij(n)} = 1 \ \wedge \ v_{j(n)} = 1) \\ &= w_{ij} v_j. \end{aligned} \tag{3.17}$$

Input summation and nonlinear transformation can be performed simultaneously using logical OR operation. The inputs of an OR gate are product sequences, $m_{ij(n)}$, produced from AND gates. Two kinds of synaptic weights w_{ij}^+ and w_{ij}^- are necessary, positive (or excitatory) and negative (or inhibitory) for most feedforward neural networks. Thus, two separate OR gates per neuron are needed to form excitatory and inhibitory net inputs. Let net_i^+ be the probability of a pulse occurrence in the output sequence $net_{i(n)}^+$ of an n-input OR gate for an excitatory net input in neuron i and net_i^- likewise for an inhibitory net input (See Figure 3.4). net_i^+ and net_i^- can be described by

$$\begin{aligned} net_i^+ &= P(net_{i(n)}^+ = 1) \\ &= P(m_{i1(n)}^+ = 1 \ \vee \ m_{i2(n)}^+ = 1 \ \vee \cdots \vee \ m_{in(n)}^+ = 1) \\ &= 1 - (1 - P(m_{i1(n)}^+ = 1))(1 - P(m_{i2(n)}^+ = 1)) \cdots (1 - P(m_{in(n)}^+ = 1)) \\ &= 1 - \prod_{j=1}^{n} (1 - m_{ij}^+) \\ &= 1 - \prod_{j=1}^{n} (1 - w_{ij}^+ v_j) \end{aligned} \tag{3.18}$$

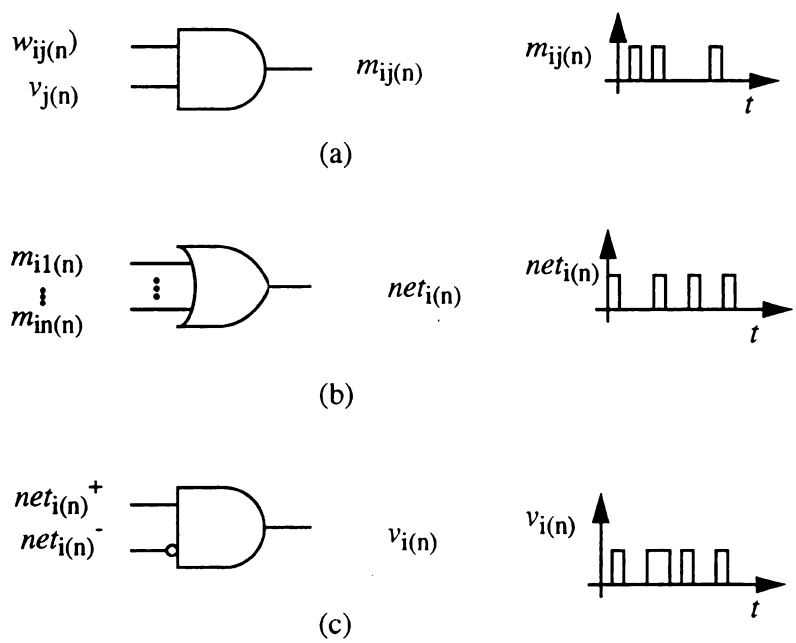


Figure 3.4. Stochastic computations in the DMNN (a) synaptic multiplication; (b) logical OR; (c) neural activation.

and

$$net_i^- = 1 - \prod_{j=1}^{n}(1 + w_{ij}^- v_j). \tag{3.19}$$

Two net inputs, formed from dedicated OR gates, AND together to form the activation function. If two sequences $net_{i(n)}^+$ and $net_{i(n)}^-$ are statistically independent, the probability of a pulse occurrence, v_i, in the activation sequence is

$$\begin{aligned} v_i &= P(v_{i(n)} = 1) \\ &= P(net_{i(n)}^+ = 1 \ \wedge \ net_{i(n)}^- = 0) \\ &= net_i^+(1 - net_i^-) \end{aligned}$$

$$= [1 - \prod_{j=1}^{n}(1 - w_{ij}^{+}v_j)] \prod_{j=1}^{n}(1 + w_{ij}^{-}v_j). \quad (3.20)$$

The nonlinear activation function, described in equation 3.20, is continuous and differentiable, indicating that back-propagation can be used for training [8]. This form of stochastic computation will be used for developing a generic DMNN architecture in the next chapter.

3.5 Back-Propagation in the DMNN

The DMNN is a feedforward neural network which can be trained with the back-propagation algorithm discussed in Section 2.1.3.2. The back-propagation algorithm performs gradient descent iteratively over a sum-squared error measure. This section shows how the non-traditional neuron activation function described in the previous section is incorporated into the back-propagation algorithm.

Define n_i as the number of neurons in the ith layer. The input layer is not counted as a layer. Accordingly, for a k-layer DMNN, n_0 and n_k indicate the number of elements in an input pattern and the number of output neurons in the output layer, respectively. The training for the DMNN can be done off-line or on-line using a digital computer. The choice depends on whether or not the resolution of the DMNN can represent the changes of synaptic weights during training for a particular application problem. The resolution of an n-bit DMNN is $\frac{1}{2^n-1}$. For example, it is approximately 10^{-3} for an 10-bit DMNN. However, more than 10^{-5} precision is often required in most application problems. That is the reason that the DMNN must be trained off-line in most cases.

Whenever an input pattern is presented to the network, the output pattern of the

output layer is compared to the target pattern; the connection weights between layers are modified in a backward direction according to the error. Given pattern μ, the sum-squared error measure is

$$E_\mu = \frac{1}{2}\sum_{i=1}^{n_k}(t_{\mu i} - v_{\mu i})^2 \tag{3.21}$$

where $t_{\mu i}$ is the target output for the ith neuron in the output layer when input pattern μ is presented and $v_{\mu i}$ is the ith element of the actual output pattern. The overall measure of the sum-squared error over p training patterns is

$$E = \sum_{\mu=1}^{p} E_\mu. \tag{3.22}$$

Thus, the back-propagation rule states that

$$\Delta w_{ij}(k) = \sum_{\mu=1}^{p} \Delta_\mu w_{ij}(k) \tag{3.23}$$

where the subscript k denotes the number of iterations and $\Delta_\mu w_{ij}$ is the change to be made to the weight from the ith to jth neuron unit following presentation of pattern μ. The gradient descent rule for positive weights states

$$\begin{aligned} \Delta_\mu w_{ij}^{+}(k) &= -\eta \frac{\partial E_\mu}{\partial w_{ij}^{+}} \\ &= -\eta \frac{\partial E_\mu}{\partial v_{\mu i}} \frac{\partial v_{\mu i}}{\partial net_{\mu i}^{+}} \frac{\partial net_{\mu i}^{+}}{\partial w_{ij}^{+}}. \end{aligned} \tag{3.24}$$

Similarly, the weight change for negative weights is given by

$$\Delta_\mu w_{ij}^{-}(k) = -\eta \frac{\partial E_\mu}{\partial v_{\mu i}} \frac{\partial v_{\mu i}}{\partial net_{\mu i}^{-}} \frac{\partial net_{\mu i}^{-}}{\partial w_{ij}^{-}}. \tag{3.25}$$

Define

$$\epsilon_{\mu i} = -\frac{\partial E_\mu}{\partial v_{\mu i}},$$

$$\delta^+_{\mu i} = -\frac{\partial E_\mu}{\partial net^+_{\mu i}} = \epsilon_{\mu i}\frac{\partial v_{\mu i}}{\partial net^+_{\mu i}},$$

$$\delta^-_{\mu i} = -\frac{\partial E_\mu}{\partial net^-_{\mu i}} = \epsilon_{\mu i}\frac{\partial v_{\mu i}}{\partial net^-_{\mu i}}. \tag{3.26}$$

By equations 3.18 to 3.20, we can obtain

$$\frac{\partial v_{\mu i}}{\partial net^+_{\mu i}} = 1 - net^-_{\mu i},$$

$$\frac{\partial net^+_{\mu i}}{\partial w^+_{ij}} = (1 - net^+_{\mu i})\frac{v_{\mu j}}{1 - w^+_{ij}v_{\mu j}}$$

and

$$\frac{\partial v_{\mu i}}{\partial net^-_{\mu i}} = -net^+_{\mu i},$$

$$\frac{\partial net^-_{\mu i}}{\partial w^-_{ij}} = -(1 - net^-_{\mu i})\frac{v_{\mu j}}{1 + w^+_{ij}v_{\mu j}}.$$

In the output layer,

$$\epsilon_{\mu i} = t_{\mu i} - v_{\mu i}. \tag{3.27}$$

In the hidden layers,

$$\epsilon_{\mu i} = -\frac{\partial E_\mu}{\partial v_{\mu i}}$$

$$= \sum_k -\frac{\partial E_\mu}{\partial net^+_{\mu k}} \frac{\partial net^+_{\mu k}}{\partial v_{\mu i}} + \sum_k -\frac{\partial E_\mu}{\partial net^-_{\mu k}} \frac{\partial net^-_{\mu k}}{\partial v_{\mu i}}$$

$$= \sum_k [\delta^+_{\mu k}(1 - net^+_{\mu k}) \frac{w^+_{ki}}{1 - w^+_{ki} v_{\mu i}}] + \sum_k [-\delta^-_{\mu k}(1 - net^-_{\mu k}) \frac{w^-_{ki}}{1 + w^-_{ki} v_{\mu i}}]. \quad (3.28)$$

Then, the changes in positive and negative weights, resulting from the presentation of training pattern μ are described respectively by following recursive forms:

$$\Delta_\mu w^+_{ij}(k) = \eta \delta^+_{\mu i} \frac{\partial net^+_{\mu i}}{\partial w^+_{ij}}$$

$$= \eta \delta^+_{\mu i}(1 - net^+_{\mu i}) \frac{v_{\mu j}}{1 - w^+_{ij} v_{\mu j}} \quad (3.29)$$

and

$$\Delta_\mu w^-_{ij}(k) = \eta \delta^-_{\mu i} \frac{\partial net^-_{\mu i}}{\partial w^-_{ij}}$$

$$= -\eta \delta^-_{\mu i}(1 - net^-_{\mu i}) \frac{v_{\mu j}}{1 + w^+_{ij} v_{\mu j}} \quad (3.30)$$

where $\delta^+_{\mu i} = \epsilon_{\mu i}(1 - net^-_{\mu i})$ and $\delta^-_{\mu i} = -\epsilon_{\mu i} net^+_{\mu i}$.

The back-propagation algorithm incorporating the activation function implemented in the DMNN has two forms:

$$\Delta w_{ij}(k) = \begin{cases} \eta \sum_{\mu=1}^{p} \delta^+_{\mu i}(1 - net^+_{\mu i}) \frac{v_j}{1 - w^+_{ij} v_j} & \text{if } w_{ij} = w^+_{ij} \\ -\eta \sum_{\mu=1}^{p} \delta^-_{\mu i}(1 - net^-_{\mu i}) \frac{v_j}{1 + w^-_{ij} v_j} & \text{if } w_{ij} = w^-_{ij}. \end{cases} \quad (3.31)$$

Gradient descent, described above, can be extremely slow for small η while it can oscillate for large η [43]. In order to achieve the most rapid learning, a learning rate η which is as large as possible without leading to oscillation must be chosen. One way to accelerate the learning is to add a *momentum term*.

$$\Delta w_{ij}(k) = -\eta \sum_{\mu=1}^{p} \frac{\partial E_\mu}{\partial w_{ij}} + \alpha \Delta w_{ij}(k-1) \quad (3.32)$$

where α is the *momentum parameter* such that $0 \leq \alpha \leq 1$. α determines the effect of past weight changes on the current direction of movement in weight space. This 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 high-frequency variations of the error surface in the weight space. In turn, the effective learning rate can be made larger without divergent oscillations occurring. A C program implementing the back-propagation in the DMNN is listed in Appendix B.

CHAPTER 4

Pulse-mode Digital Multilayer Neural Networks

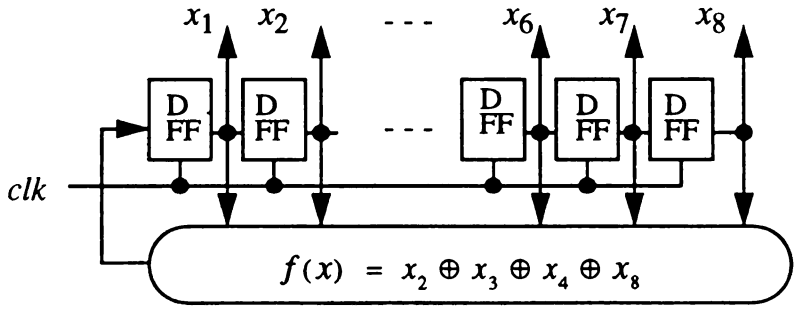
In this chapter, digital architectures of basic elements such as synaptic elments and neuron body elements are developed. Using these basic elements, the modular architecture for digital feedforward neural networks is developed as a Digital Multilayer Neural Network (DMNN). Use of simple logic gates as computing elements and modular design techniques will lead to the DMNN architecture being relatively compact in size and expandable to any size network. Furthermore, massive parallelism embedded in stochastic computations using random pulse streams is fully utilized with this architecture. A generic architecture of a DMNN coprocessor is also presented. All components in the DMNN and the DMNN coprocessor are modeled and simulated in VHDL. Use of VHDL as the modeling tool for the DMNN coprocessor is discussed briefly. Finally, the hardware complexity of the DMNN is estimated.

4.1 Basic Computing Elements

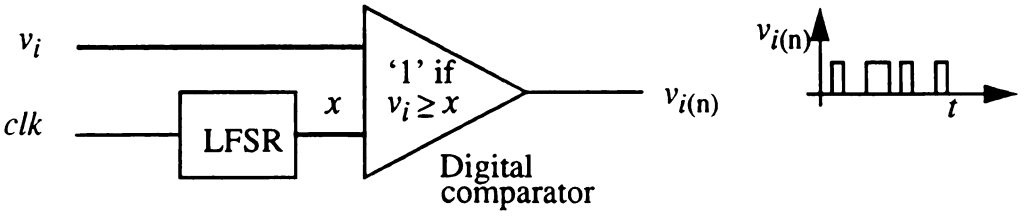
A random pulse generator, a synaptic element, an input neuron body element, and a regular neuron body element are developed as basic computing elements in the DMNN. These basic elements are used to develop a modular network architecture.

4.1.1 Random Pulse Generator

The block diagram of a random pulse generator was presented in Chapter 3. The random pulse generator (RPG) is comprised of a tapped LFSR and a digital comparator. In Figure 4.1(a), the order of a LFSR is 8 and the example feedback function is $f(x) = x_2 \oplus x_3 \oplus x_4 \oplus x_8$ implemented by XOR logic gates, where the period of sequence $v_{(n)}$ is $2^8 - 1 = 255$. Figure 4.1(b) shows the structure of a random pulse generator using D flip-flops, XOR logic gates, and a digital comparator.



(a)



(b)

Figure 4.1. (a) A maximum length 8-order LFSR where $f(x) = x_2 \oplus x_3 \oplus x_4 \oplus x_8$; (b) a pseudo-random pulse generator for v_i.

At every clock period, a logic '1' pulse is generated if $v_i \geq x$. Otherwise, a logic '0' pulse is generated.

4.1.2 Synaptic Element

A large number of synaptic multiplications are required, even for a small size feedforward neural network. For example, if the network consists of m layers excluding an input layer, the number of synaptic multiplications required per feedforward operation is

$$\sum_{l=1}^{m} n_l n_{l-1}$$

where n_l is the number of neuron elements in the lth layer and n_0 is the number of input patterns applied to the input layer. Each synaptic multiplication in the DMNN is performed relatively more slowly than a deterministic calculation done on a digital computer, but all the multiplications in the network can be performed in parallel.

Let w_{ij} and v_j be the synaptic weight between neuron elements i and j and the neural activation in neuron element i, respectively. Figure 4.2 shows the structure and block diagram of a digital synaptic element (SYN). The VHDL code for a SYN model is listed in Appendix C. The SYN consists of a random pulse generator (RPG), a weight register, two AND gates, and two wired-OR lines. Weight w_{ij} is represented as an r-bit fractional number, where the MSB is a sign bit and the rest represent the magnitude in sign-magnitude format. With w_{ij} loaded into a weight register, the corresponding random pulse stream $w_{ij(n)}$ is generated through the RPG. The pulse stream is transmitted to two AND gates: the upper one for positive weights and the lower one for 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.

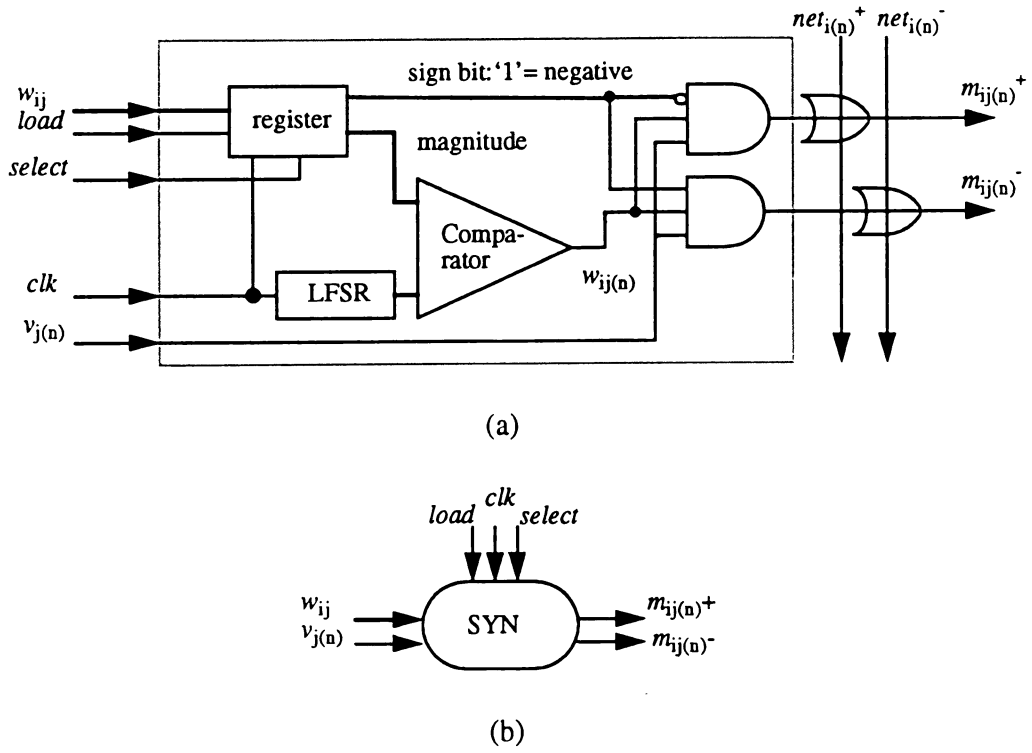


Figure 4.2. (a) A synaptic element (SYN); (b) a block diagram of a SYN.

4.1.3 Input Neuron Body Element

An input neuron body element (INB) consists of an n-bit register, a tapped LFSR, and a digital comparator. Figure 4.3 shows the structure and block diagram of the INB. The tapped LFSR and the digital comparator forms a random pulse generator (RPG). The role of the INB is to convert the value of the ith element in an input pattern, v_i, to a corresponding random pulse sequence $v_{i(n)}$. No computation occurs in this element. v_i is loaded into the register at the rising edge of the clock with *load* = '1' and *select* = '1'. The *select* signal corresponds to the word line from an address decoder. A binary pulse $v_{i(n)}$ is generated every clock cycle when *load* = '0'.

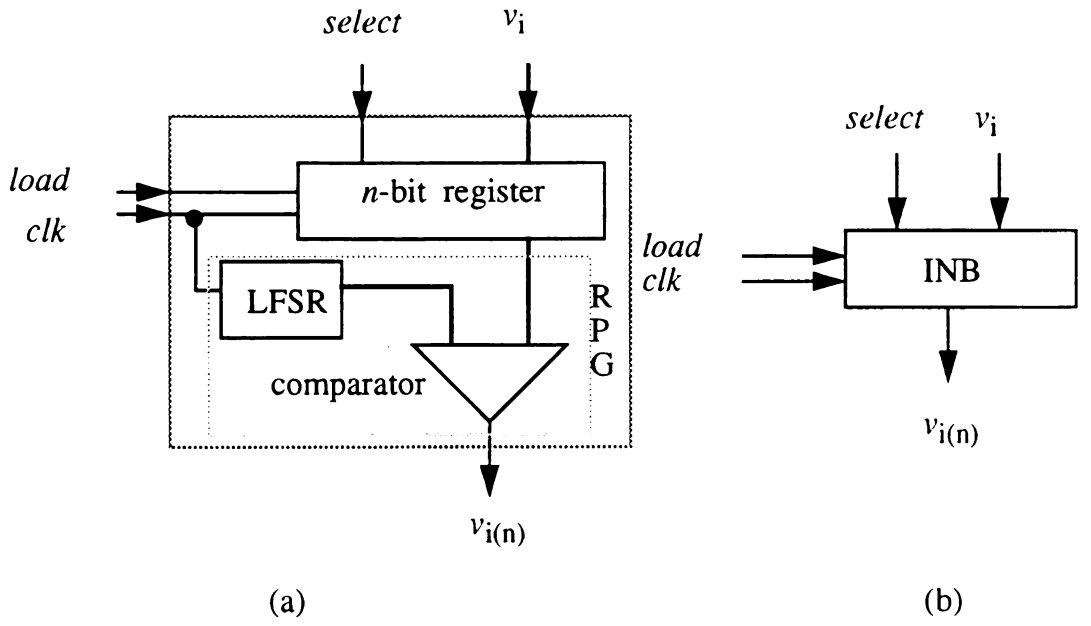


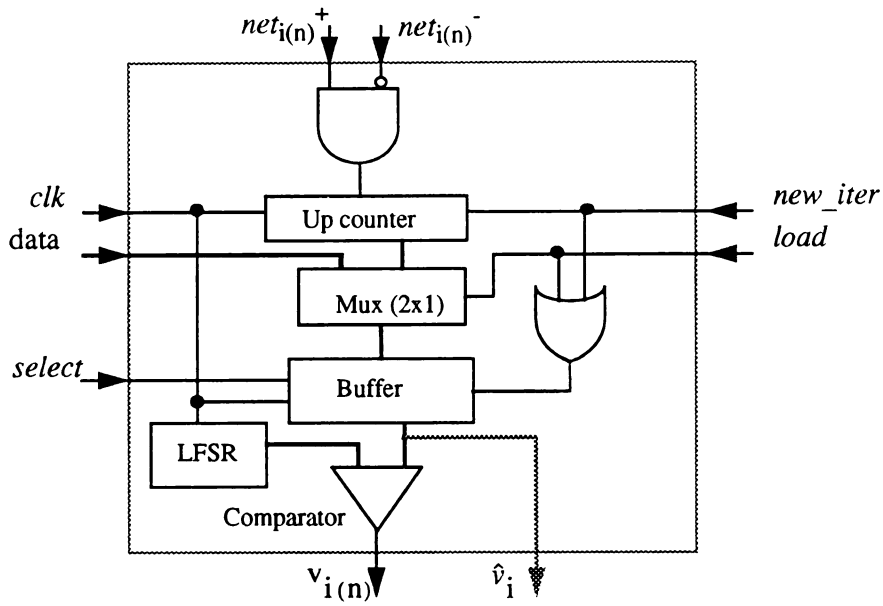
Figure 4.3 . (a) An input neuron body (INB); (b) a block diagram of INB.

4.1.4 Regular Neuron Body Element

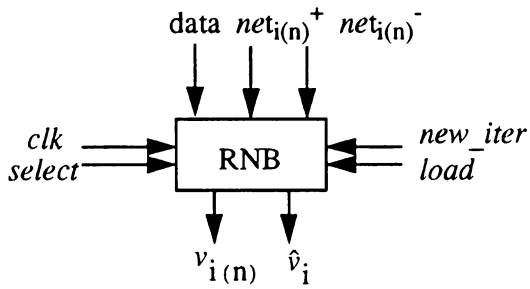
Two net-input pulse streams, transmitted from synaptic elements, are collected in an up-counter in a regular neuron body element (RNB) through an AND gate to form a neural activation. Figure 4.4 shows the structure and block diagram of an RNB. A VHDL model of the RNB is listed in Appendix C. An RNB consists of an AND gate, an OR gate, an up-counter, 2x1 multiplexers, a buffer, and an RPG. Let net_i^+ and net_i^- be an excitatory and inhibitory input for neuron i, respectively. In a DMNN, product sequences $m_{ij(n)}$ from synaptic elements are logically ORed to form a net-input. $net_{i(n)}^+$ is ANDed with $(net_{i(n)}^-)'$ to form a neural activation v_i in neuron i, where $(net_{i(n)}^-)'$ is the complement of $net_{i(n)}^-$. v_i is estimated as $\hat{v}_i$ which is actually the value of the up-counter after each iteration.

After each iteration, the signal *new_iter* changes '0' to '1' and then the output of a counter is transferred to a buffer via a 2x1 multiplexer at the next clock. At a 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. v_i (dotted arrow) is used as an output of a neuron i in the output layer, while $v_{i(n)}$ (solid arrow) is used in the hidden layers. If *load* is '1', the buffer is reset to an initial input value at a new clock cycle. Otherwise, a new neuron state is loaded when *new_iter* is '1'. The buffer is enabled when *load* or *new_iter* is '1'.



(a)



(b)

Figure 4.4. (a) A regular neuron body element (RNB); (b) a block diagram of RNB.

4.2 Modular Architecture

The DMNN can be constructed from four basic modules: input layer module, synaptic array module, regular neuron body array module, interconnection module. The input layer module (ILM) is composed of a group of input neuron body elements. It receives inputs and transforms them into corresponding binary pulse sequences. Figure 4.5 shows the structure of the ILM. The pulse sequences generated are transmitted to the synaptic elements in the next layer through an interconnection module (ICM). A synaptic array module (SAM) consists of a group of synaptic elements and net-input lines. Figure 4.6 shows the structure of the SAM. Synaptic weights w_{ij} are loaded before network operations start. $w_{ij(n)}$'s from SYNs in the SAM are logically ANDed with $v_{j(n)}$ transmitted from the previous layer. All synaptic multiplications in the same layer are performed simultaneously. A regular neuron body array module (RNAM) consists of a group of regular neuron bodies. Figure 4.7 shows the structure of RNAM. Pulses on two net-input sequences transmitted from the SAM are collected in an up-counter of neuron i through an AND gate to form $v_{i(n)}$. All $v_{i(n)}$'s from the RNAM are produced simultaneously.

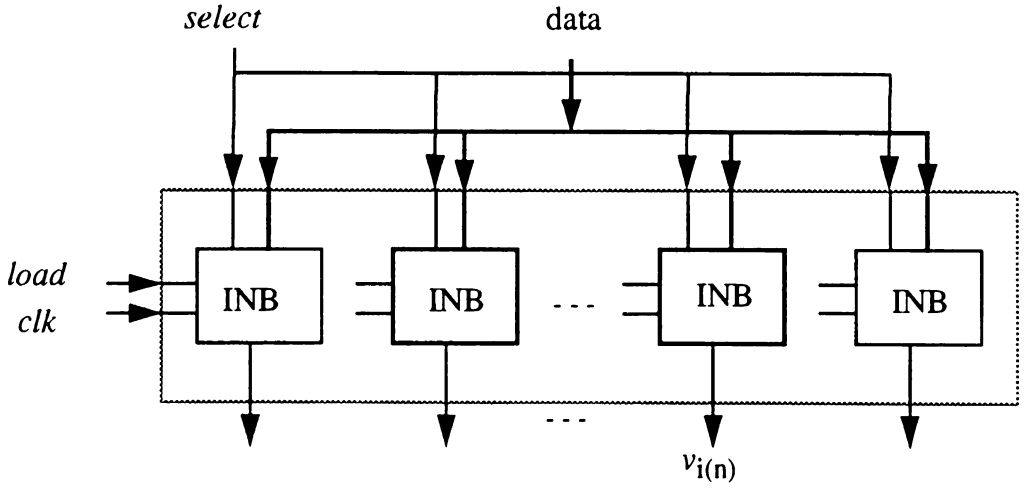


Figure 4.5. An input layer module (ILM).

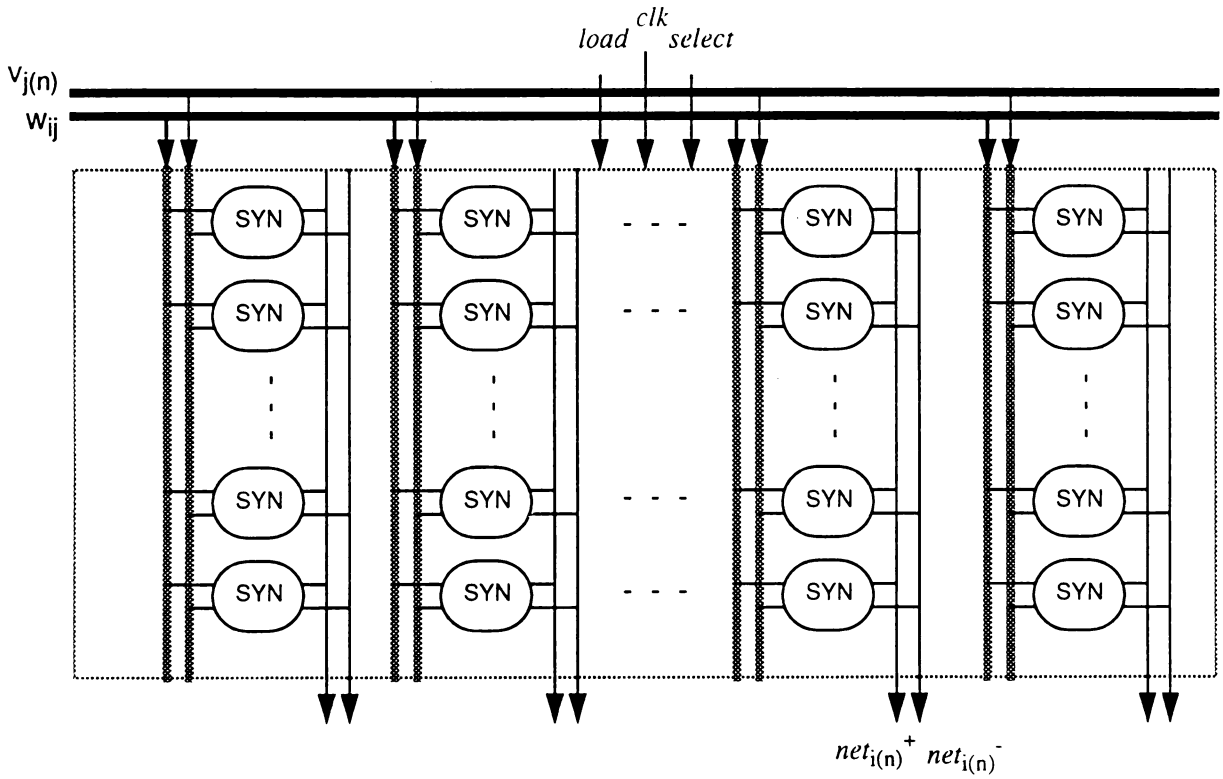


Figure 4.6. A synaptic array module (SAM).

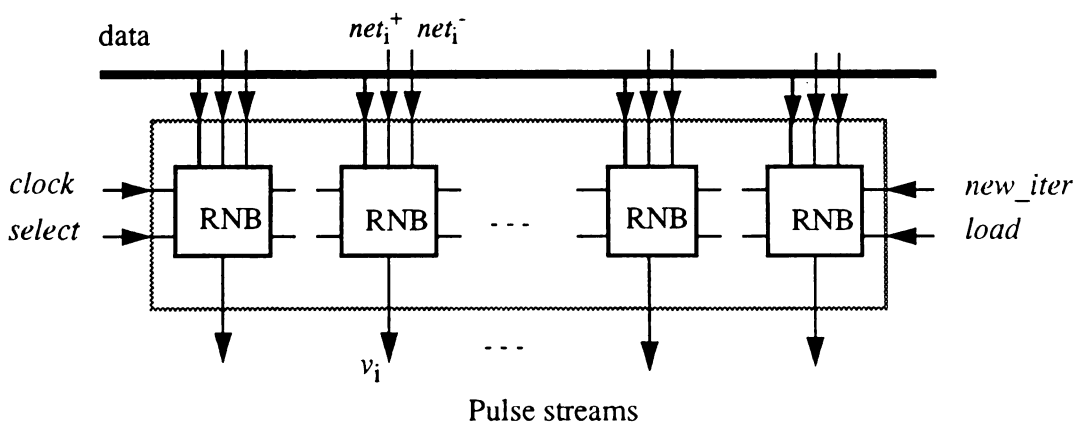


Figure 4.7. A regular neuron body array module (RNAM).

Since the action pulse sequences from the previous layer are used to form the action pulse sequences in the neurons of the next layer, there may exist a correlation between new action pulse sequences in the next layer without the buffering scheme used in the RNBs. The dedicated pulse generator in each RNB functions as a filter eliminating correlation by producing the uncorrelated pulse sequences for the new neuron activations formed after each iteration. Again, an ICM is a group of connection lines that transmit action pulse sequences from the previous layer to the SAM in the next layer.

Using the modules discussed above, any size DMNN with an arbitrary number of neurons in each layer and an arbitrary number of the layers can be configured. Figure 4.8 shows the general architecture of the DMNN. Just like in a pipelined architecture, once the layers are full, results from the output layer are available after each sampling clock period.

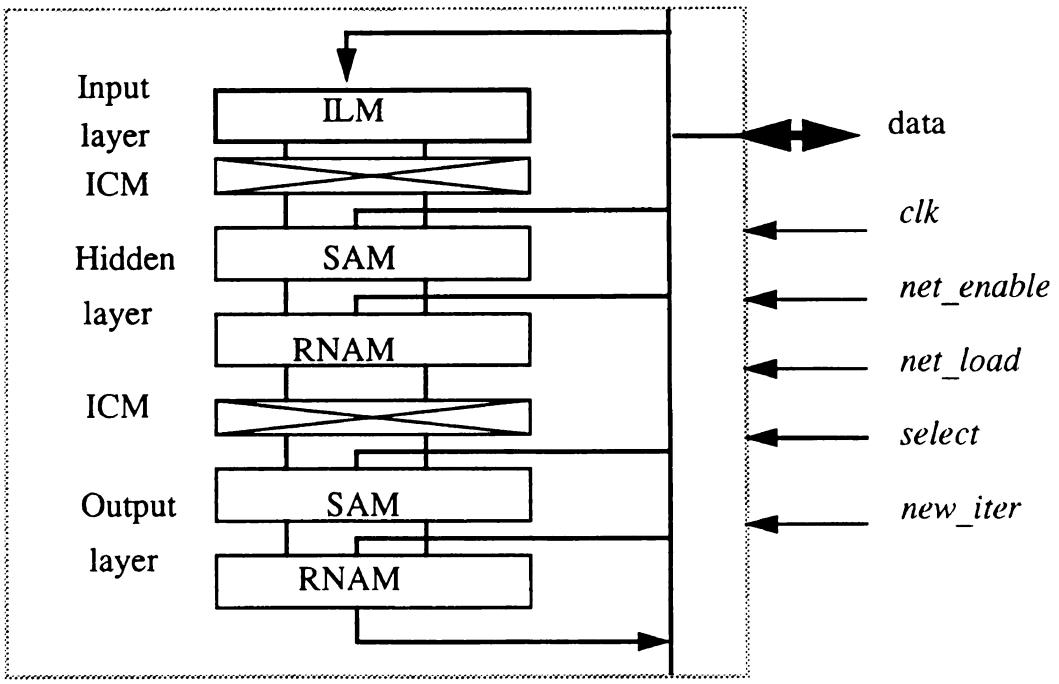


Figure 4.8. The general architecure of the DMNN.

An advantage of this architecture is that there are no unused synaptic elements in the network and there is no n-bit prescaler in a neuron body. This increases the possible neuron-density on a chip while retaining the high degree of flexibility and expandability. The DMNN network can be optimized in terms of the minimum number of neurons in a given number of layers when it is customized for a particular application.

4.3 DMNN Coprocessor

The DMNN can be viewed as a coprocessor as shown in Figure 4.9. The coprocessor is composed of a DMNN, a controller, a memory, an iteration counter (IRC), and a clock generator. The DMNN coprocessor can be attached to a host computer

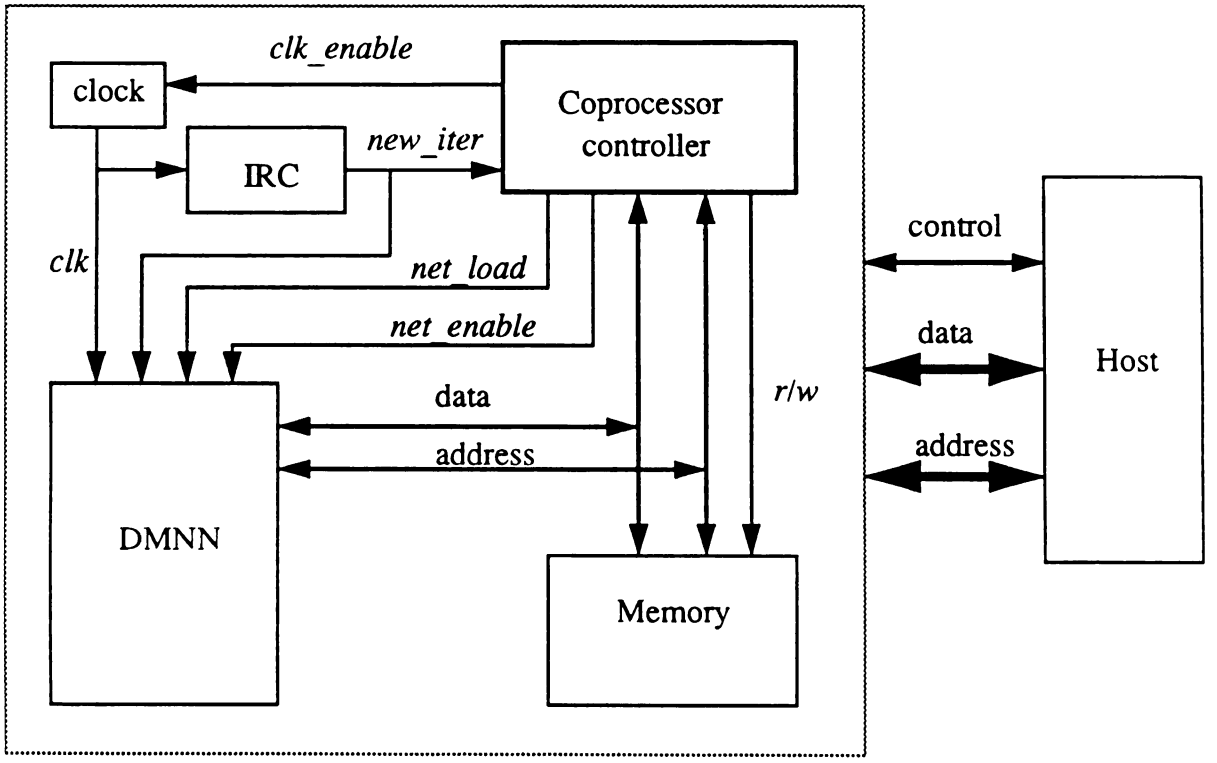


Figure 4.9. A DMNN coprocessor.

through an standard interface (not shown). The controller consists of a microprocessor and a control unit which can be microprogrammed or hardwired. The memory and the DMNN in the coprocessor may have their own address decoders. The network is trained on the host computer. After training, the network configuration, trained weights, input patterns, and some control commands are downloaded from the host memory.

4.4 Behavioral model of a DMNN Coprocessor

The DMNN architecture proposed in the preceding sections is modeled using VHDL. The design methodology and procedure for developing a DMNN coprocessor are discussed here. A DMNN coprocessor and a DMNN architecture are used as example behavioral models using VHDL in this section. Complete VHDL code listings for all other components required to implement the coprocessor can be found in Appendix C.

4.4.1 Introduction

Functional behavior and structure of the DMNN can be modeled and simulated using VHDL. Various advantages are obtained by using VHDL as a modeling tool for the DMNN architecture. These advantages include:

1. The function and logical organization of the DMNN can be developed and tested without involving details of the implementing technology.

2. Design modification resulting from design errors or changes can be made in an early stage of design process without additional cost.

3. VHDL is flexible, allowing different network configurations to be modeled without significant changes to an original design.

4.4.2 Design Methodology

A top-down design approach is needed to manage the complexity of large systems like a DMNN. The model of the DMNN coprocessor starts with a high-level specification of the network. The high level description is decomposed into lower level specifications in a hierarchical fashion. Figure 4.10 shows the design hierarchy of a DMNN coprocessor. At the highest level, the whole system can be viewed as a coprocessor. This coprocessor is built using a DMNN, a control unit, a clock generator, an iteration generator, and memory.

Each component (or VHDL design entity), shown in Figure 4.10, is described in VHDL using either of two styles of descriptions: behavioral or structural. All components (dotted boxes or ovals) residing in lower branches of the hierarchy are modeled using behavioral descriptions. The reason for this is that the structures or gate level designs of these components are well known and available in most design automation (DA) libraries. Secondly, the complete structural descriptions for these components cause the resulting VHDL simulation kernel to run extremely slow and to occupy a large amount of memory. Each behavioral description can be eventually replaced by a structural description without changing other entities. All other components are modeled using structural descriptions.

This hierarchical design scheme combined with the natural modularity of the DMNN architecture makes design modification and simulation much easier.

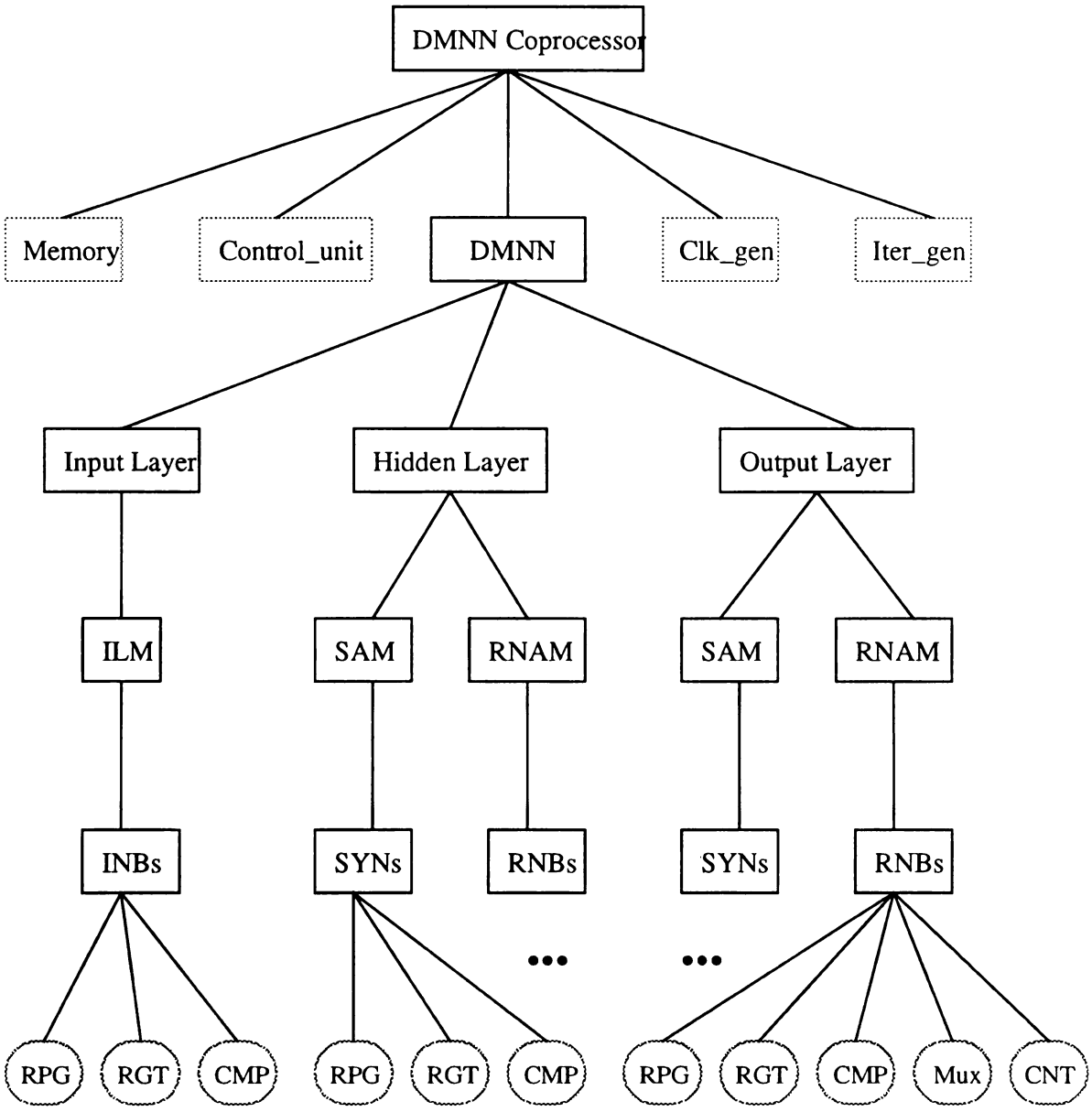


Figure 4.10. The design hierarchy of a DMNN coprocessor

4.4.3 Coprocessor Control in VHDL

In order to model the DMNN coprocessor in VHDL and apply the model to some example problems, the following procedures have been followed.

Step 1. Back-propagation for the DMNN, described in Chapter 3, trains the network, finds a network architecture converging below a predefined sum-squared error, and then writes the final synaptic weights and

Step 2. The DMNN coprocessor reads these files and a file for test patterns.

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

Step 4. The coprocessor initiates the controller, and the controller starts generating control signals.

Step 5. The network (DMNN) starts classifying input patterns until classifications are completed. The output patterns are written into memory each time one input pattern is classified.

Step 6. Results are written into a file.

A part of VHDL code for the DMNN coprocessor is shown as a test bench in Figure 4.11. The complete code is listed in Appendix C. The network architecture, problem specification, variable definition, and some subprograms are described in a package, named *dmnn_pack*. Once the *run* signal changes from '0' to '1', a simulation kernel autonomously performs Step 2 to Step 6 as above.

```
use work.dmnn_pack.all;
entity dmnn_test_bench is
end dmnn_test_bench;

architecture behavior of dmnn_test_bench is
component clock
        port(start: in bit;
             clk_out: out bit);
end component;
component step_cnt          - - -        end component;
component dmnn_net           - - -         end component;
component control_unit      - - -        end component;
component RAM                - - -         end component;

type out_bits is array(natural range 1 to out_unit_no) of bit;
signal run, clk_sig,  net_load: bit:='0';
signal net_ena_sig, clk_ena,  new_iter: bit:='0';
signal pattern_sig: neural_array;  signal weight_sig: neural_matrix;
signal state_sig: out_state_array;
signal ram_rw, ram_ena:  bit;
signal ram_net_state: pattern_matrix;

for all: clock use entity work.clock(clk_behavior);
for all: step_cnt use entity work.step_cnt(step_cnt_dmnn);
for all: control_unit use entity work.control_unit(behavior);
for all: dmnn_net use entity work.dmnn_net(behavior);
for all: RAM use entity work.RAM(behavior);

 begin          run <= '1' after 5 ns;
                Clk_Blk: clock port map (clk_ena, clk_sig);
                Step_Blk: step_cnt port map (clk_sig, new_iter);
          Control_Blk: control_unit port map (run,new_iter,net_load, ram_rw,
                                              clk_ena,net_ena_sig,ram_ena);
             dmnn_net_BLK: dmnn_net port map (net_load, clk_sig, net_ena_sig,
                                        new_iter,pattern_sig,weight_sig,state_sig);
               RAM_BLK: RAM port map (ram_ena,ram_rw,state_sig,
                                                  pattern_sig,weight_sig);
end behavior;
```

Figure 4.11. VHDL code implementing the DMNN coprocessor.

4.4.4 DMNN Model in VHDL

The DMNN itself consists of an input layer, a hidden layer(s), and an output layer. Each layer and all components required in the layer are generated by component instances in structural descriptions. Figure 4.12 shows the part of VHDL code for the DMNN model. Any size network can be modeled by specifying the network configuration in the *dmnn_pack*. The components are connected by wires, modeled as signals in VHDL. The network supports complete connectivity between neuron elements, but unnecessary synaptic elements are effectively disconnected by loading a zero weight for that synapse.

```
use work.dmnn_pack.all;
entity dmnn_net is port ( net_load, net_clk, net_enable, next_period: in bit;
                          new_pattern: in neural_array; weight_in: in neural_matrix;
                          net_state_out: out out_state_array);
end dmnn_net;

architecture behavior of dmnn_net is

    component synapse  - - -  end component;
    component neuron_body  - - -  end component;
    signal clk: bit; signal syn_ran_in: neural_matrix;
    signal body_ran_in, net_output: neural_array; - - -
    for all: synapse use entity work.synapse(synap_dmnn);
    for all: neuron_body use entity work.neuron_body(nbody_dmnn);

begin  Main_blk: block(net_enable='1') begin
                 clk<= guarded net_enable and net_clk;
                 ex_wired_or<=guarded Wired_Or(ex_or);
                 in_wired_or<=guarded Wired_Or(in_or);
       end block;
       Input_Layer: - - -
       Hidden_Layer: - - -
       Output_Layer: - - -
end behavior;
```

Figure 4.12. VHDL code implementing the DMNN.

4.5 Hardware Complexity

To access the hardware complexity of the DMNN, define A_g and $A_{(X)}$ as the chip areas required by an n-input NAND gate and by component X, respectively. For a reasonable number of inputs, say $n \leq 10$, it is assumed that an inverter is viewed as a special case of an n-input NAND and an n-input AND is modeled as an n-input NAND followed by an inverter. If all components and modules in the DMNN are designed only using NAND gates and necessary connections, a measure of the hardware complexity of a DMNN with m layers is given by

$$A_{(DMNN)} = A_{(INB)} \cdot n_0 + A_{(RNB)} \sum_{i=1}^{m} n_i + A_{(SYN)} \sum_{i=1}^{m} n_i \cdot n_{i-1} \tag{4.1}$$

where n_0 and n_i are the number of elements in an input pattern applied to the network and the number of neurons in the ith layer, respectively.

Assume that a D or J-K flip flop (FF) is implemented by 5 NAND gates ($A_{(FF)} = 5A_g$), a 2x1 multiplexer (MUX) by 4 NAND gates ($A_{(MUX)} = 4A_g$), and a two-input exclusive OR (XOR) gate by 4 NAND gates ($A_{(XOR)} = 4A_g$). The gate level schematics of essential basic digital components, such as an n-bit register, an n-bit comparator, and an n-bit up-counter, can be found in Appendix D. Table 4.1 presents the relative chip area required by each of these components based on the unit NAND gate.

In Table 4.1, it is assumed that 3 XOR gates are needed to implement a feedback function of a LFSR. This leads to the assessment of hardware complexity for the INB, RNB, and RNB modules. Table 4.2 shows the chip area required by these DMNN elements.

Using equation 4.1 and Table 4.2, the chip area required by any size DMNN can be estimated. Table 4.3 presents the chip area of two example networks with respect

to the register length n and the unit NAND gate area A_g. A network configuration in this table is represented by n_0 x n_1 x ..., where n_0 and n_i denote the number of neurons in the input layer and the number of neurons in the ith layer, respectively.

Table 4.1. Chip area required by basic digital components.

Component	Chip area
n-bit register (RGT)	$nA_{(FF)} + 2nA_g = 7nA_g$
n-order LFSR	$nA_{(FF)} + 3A_{(XOR)} = (5n + 12)A_g$
n 2x1 MUXs (MUXs)	$nA_{(MUX)} = 4nA_g$
n-bit comparator (CMP)	$6nA_g + 2A_g = (6n + 2)A_g$
n-bit up-counter (CNT)	$nA_{(FF)} + 2(n-1)A_g = (7n - 2)A_g$

Table 4.2. Chip area required by DMNN elements.

DMNN element	Chip area
INB	$A_{(RGT)} + A_{(LFSR)} + A_{(CMP)} = (18n + 14)A_g$
RNB	$A_{(INB)} + A_{(CNT)} + A_{(MUXs)} + 5A_g = (29n + 17)A_g$
SYN	$A_{(INB)} + 5A_g = (18n + 19)A_g$

Table 4.3. Chip area required by two example networks.

Register length (n)	Network configuration	
	36 x 9 x 10	36 x 30 x 10
8	77,901 A_g	226,788 A_g
9	86,554 A_g	253,436 A_g
10	95,203 A_g	280,084 A_g

CHAPTER 5

Analysis of the DMNN

The statistical model of the estimated probability generated from the random pulse generator was developed in Chapter 3. Based on this model, statistical models of synaptic multiplication and signal integration are developed. Then the models are extended to hidden layers and the output layer. An analysis which predicts the error bounds on the results of the computations occurring in the DMNN is also presented. A critical comparison to deterministically computed results can then be made.

5.1 Statistical Models

For analysis, it is assumed that multiple pulse sequences are statistically uncorrelated and the effect of quantization is negligible when the register length is greater than 7 [67, 78].

5.1.1 Synaptic Multiplication

An algebraic expression for the output of an AND gate in a synaptic element is $m_{ij} = w_{ij}v_j$. Since the estimations $\hat{w_{ij}}$ and $\hat{v_j}$ of w_{ij} and v_j can be represented by the original signal plus random noise (error) in the first hidden layer, the estimation $\hat{m_{ij}}$ of m_{ij} can be described by

$$\begin{aligned} \hat{m_{ij}} &= (w_{ij} + \Delta_{\hat{w_{ij}}})(v_j + \Delta_{\hat{v_j}}) \\ &= w_{ij}v_j + w_{ij}\Delta_{\hat{v_j}} + v_j\Delta_{\hat{w_{ij}}} + \Delta_{\hat{w_{ij}}}\Delta_{\hat{v_j}} \end{aligned} \tag{5.1}$$

where $\Delta_{\hat{x}}$ is random noise with zero mean and variance $Var(\hat{x})$. The expected value of $\hat{m_{ij}}$ is

$$E(\hat{m_{ij}}) = m_{ij} = w_{ij} \cdot v_j \tag{5.2}$$

and the variance of $\hat{m_{ij}}$ is

$$Var(\hat{m_{ij}}) = w_i^2 \cdot Var(\hat{v_j}) + v_j^2 \cdot Var(\hat{w_{ij}}) \tag{5.3}$$

where the second order term is omitted from equation 5.1. In reality, the second order term is not zero because two pulse sequences have a very small, but still negligible, cross-correlation [67].

5.1.2 Two-input Logical OR

Before developing the statistic model of an n-input logical OR, the model for a 2-input OR is developed. Let $m_{i1} = m_{i(max)} = max[m_{i1}, m_{i2}]$ and let n_k be the number of neurons in the kth hidden layer. The statistic model for $\hat{net_i}$ is complex

because the output estimation results from the nonlinear transformation of $\hat{m_{ij}}$'s. It can be approximated, however, using the hypergeometric distribution. The key reason for using this model is that the deterministic nature can be observed in the output sequence, as shown in dotted lines in Figure 5.1.

The pulses of $m_{i1(n)}$ always appear on the output sequence $net_{i(n)}$. When $m_{i1} = l_{i1}/N$ and $m_2 = l_{i2}/N$,

$$NET_i = l_{i1} + W,$$

where W is a random variable indicating the number of pulse occurrences in $m_{i2(n)}$ during $N - l_{i1}$ clock cycles and NET_i (with capital letters) represents the number of pulses in $net_{i(n)}$ observed during the sampling period.

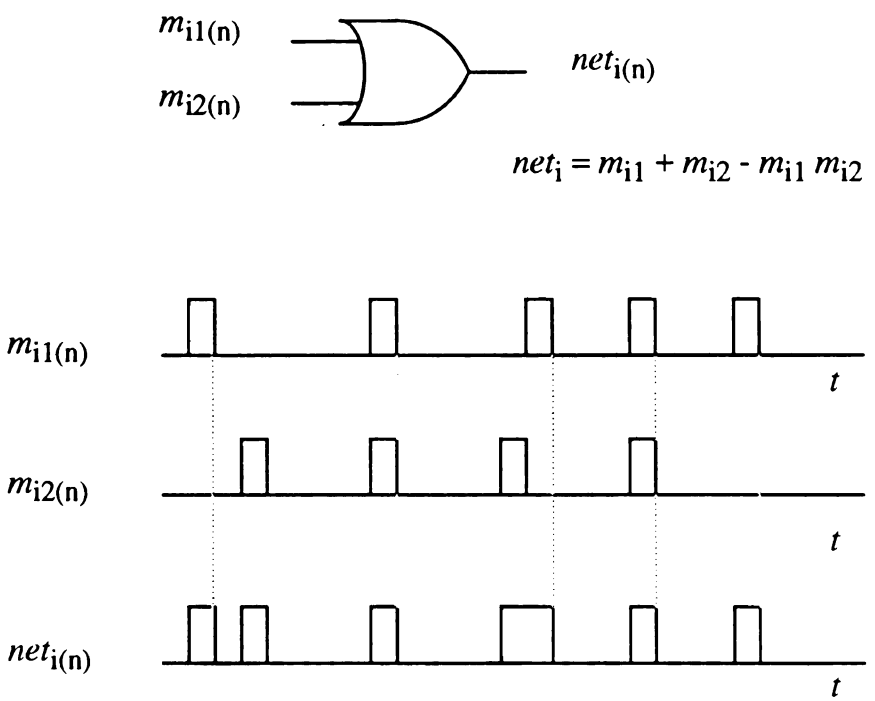


Figure 5.1. 2-input logical OR where the dotted lines illustrate the deterministic nature of the output sequence $net_{i(n)}$.

The probability function of W is

$$P(W=k) = \frac{\binom{l_{i2}}{k}\binom{N-l_{i2}}{N-l_{i1}-k}}{\binom{N}{N-l_{i1}}}.$$

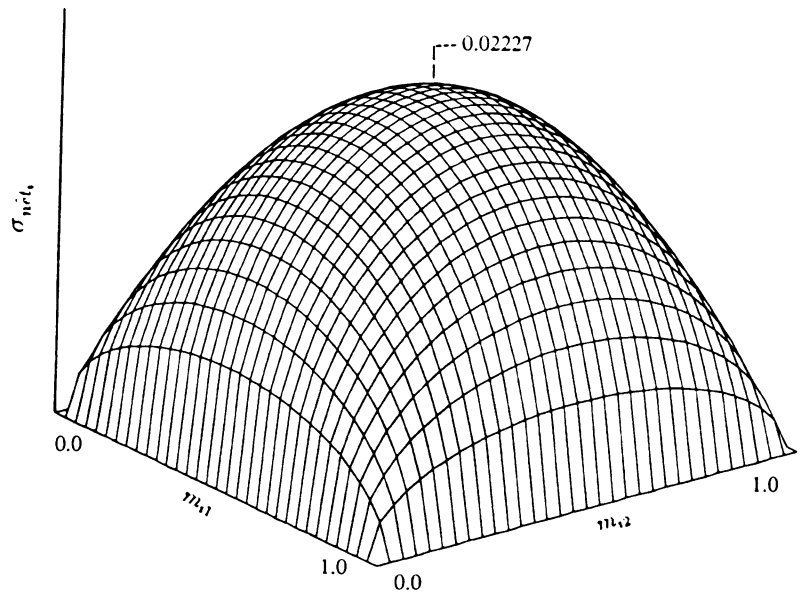
The expected value of $\hat{net}_i$ is

$$\begin{aligned} E(\hat{net}_i) &= \frac{l_{i1}}{N} + \frac{1}{N}\sum_{k=0}^{N} k \cdot P(W=k) \\ &= \frac{l_{i1}}{N} + \frac{(N-l_{i1})l_{i2}}{N^2} \\ &= m_{i1} + m_{i2} - m_{i1}m_{i2} \\ &= 1 - \prod_{j=1}^{2}(1-m_{ij}). \end{aligned} \tag{5.4}$$

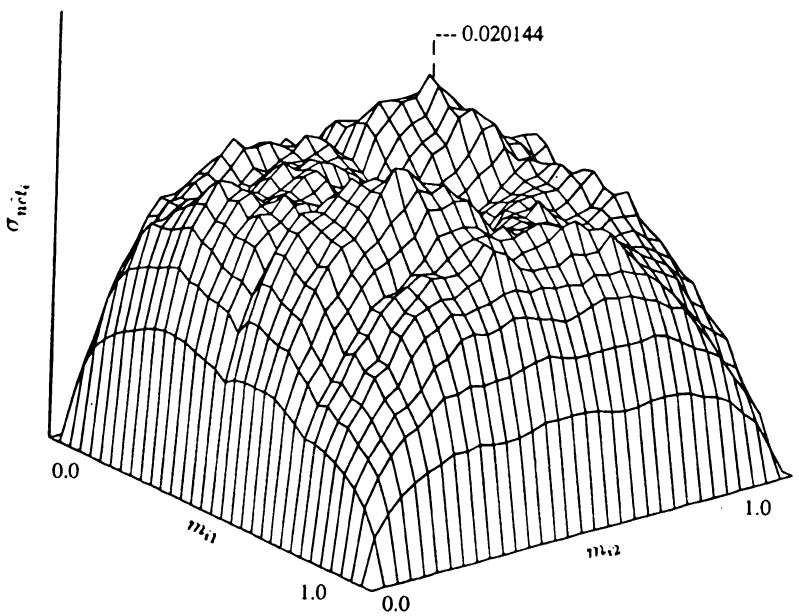
The variance of $\hat{net}_i$ is

$$\begin{aligned} Var(\hat{net}_i) &= \frac{1}{N^2}Var(W) \\ &= \frac{N-l_{i1}}{N^2}m_{i2}(1-m_{i2})\frac{l_{i1}}{N-1}. \end{aligned} \tag{5.5}$$

Figure 5.2 shows the standard deviation, $\sigma_(\hat{net}_i) = \sqrt{Var(\hat{net}_i)}$, obtained from equation 5.5 (a,d) and those obtained by simulation (b,c,e,f). In this figure, $N = 127$ for (a) to (c) and $N = 255$ for (d) to (f). The same feedback functions with different LFSR initial values for m_{i1} and m_{i2} are used in (b) and (e) while different polynomial functions are used in (c) and (f). $x_1 \oplus x_2 \oplus x_3 \oplus x_4$ is used as a feedback function for LFSRs for m_{i1} and m_{i2} in (b) and (e). $x_1 \oplus x_2 \oplus x_3 \oplus x_4$ and $x_2 \oplus x_3 \oplus x_4 \oplus x_5$ are used for m_{i1} and m_{i2} in (c) and (f), respectively.

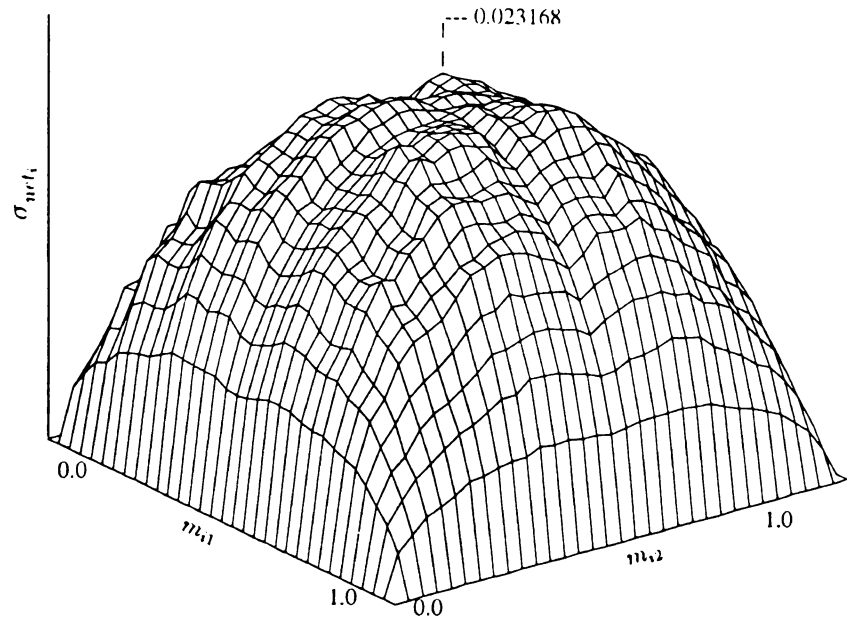


(a)

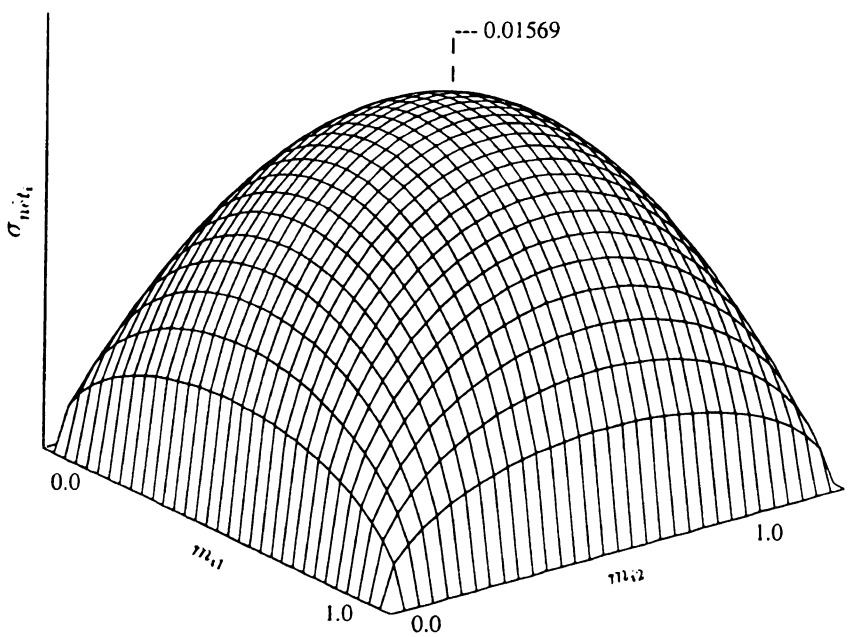


(b)

Figure 5.2. $\sigma_{n\hat{e}t_i}$ obtained from equation 5.5 when (a) $N = 127$; (d) $N = 255$, and $\sigma_{n\hat{e}t_i}$ obtained from actual simulations when (b,c) $N = 127$; (e,f) $N = 255$.

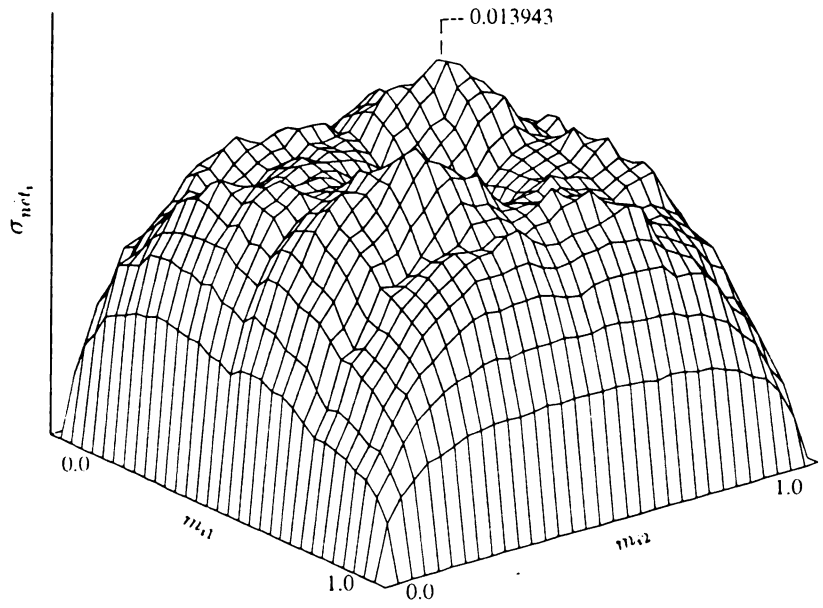


(c)

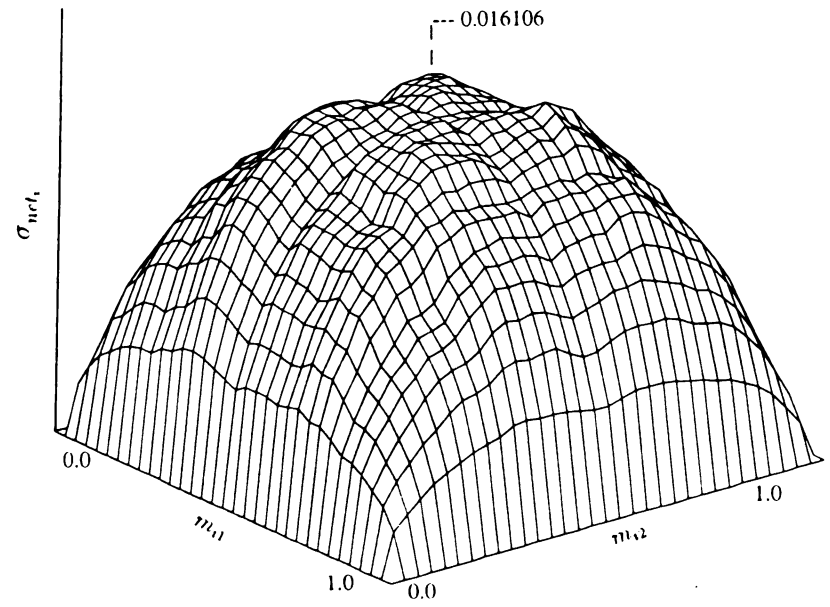


(d)

Figure 5.2. Continued.



(e)



(f)

Figure 5.2. Continued.

5.2 Effects of Random Noise in Hidden Layers

5.2.1 First Hidden Layer

In the following, assume $P_s = n = N$, as is the case in the DMNN. This assumption eliminates any error associated with the conversion of a synaptic weight w_{ij} and a neuron activation v_j into the average pulse rates of the corresponding pulse sequences $w_{ij(n)}$ and $v_{j(n)}$. Since $Var(\hat{v_j}) = Var(\hat{w_{ij}}) = 0$ from equation 3.13 when $n = N$, then $Var(\hat{m_{ij}}) = 0$ in equation 5.3. This deterministic nature of the synaptic multiplications in the first hidden layer contributes to the high accuracy of the stochastic computing technique.

Two assumptions are made for an n-input OR. The validity of these assumptions has been justified by experimental results. Define N^k_{eff} as the effective period of an LFSR in the kth hidden layer. $N^0_{eff}(= N)$ is the effective period of an LFSR in the input layer.

Assumption 1: An increase in the number of OR inputs in the first hidden layer has the same effect as increasing the period N of the LFSR to N^1_{eff} with the sampling period, $P_s = N$, unchanged, and holding the ratio l_{ij}/N such that $m_{ij} = l_{ij}/N = (m_{ij}N^1_{eff})/N^1_{eff}$.

Assumption 2: As n increases, the deterministic ratio in net_i decreases such that $m_{i(max)} = m_{i1} = max[m_{i1}, m_{i2}, \ldots, m_{in}] = \beta\frac{net_i}{\sqrt{n_0}}$ for a fixed net_i, where $0 < \beta \leq 1$.

The ratio, N^1_{eff}/N, and β depend on the application problem, the network architecture, and the input patterns. It is impossible to find N^1_{eff} and β in closed forms. Based on our experimental results for binary classification, however, N^1_{eff} can be approximated as $N + \alpha_1(n_0 - 2)N$ for $n_0 \geq 2$ where α_1 is an incremental parameter for the first hidden layer and n_0 is the number of inputs in the input layer. α_1 varies

between 0.03 and 0.04. In the following, $\beta = 1$ is assumed.

Based on these assumptions, equations 5.4 and 5.5 for the 2-input OR can be generalized for an n_0-input OR in the first hidden layer. To do so, separate the n_0 m_{ij}'s into two quantities: $m_{i1} = m_{i(max)} = \frac{l_{i1}}{N}$ and $m_{ir} = 1 - \prod_{j=2}^{n_0}(1 - m_{ij}) = m_{ir}N^1_{eff}/N^1_{eff}$. Now $E(\hat{net}_i)$ and $Var(\hat{net}_i)$ can be obtained in a straightforward manner. NET_i is the summation of a constant l_{i1} and a random variable W,

$$NET_i = l_{i1} + W.$$

The probability function of W for an n-input OR operation is

$$P(W = k) = \frac{\binom{m_{ir}N^1_{eff}}{k}\binom{N^1_{eff} - m_{ir}N^1_{eff}}{N - l_{i1} - k}}{\binom{N^1_{eff}}{N - l_{i1}}}.$$

Thus, the expected value of $\hat{net}_i$ is

$$\begin{aligned} E(\hat{net}_i) &= \frac{l_{i1}}{N} + \frac{1}{N}\sum_{k=0}^{N} k \cdot P(W = k) \\ &= \frac{l_{i1}}{N} + \frac{(N - l_{i1})l_{ir}}{N^2} \\ &= 1 - (1 - m_{i1})\prod_{j=2}^{n_0}(1 - m_{ij}) \\ &= 1 - \prod_{j=1}^{n_0}(1 - m_{ij}). \end{aligned} \tag{5.6}$$

The variance of $\hat{net}_i$ is

$$Var(\hat{net}_i) = \frac{(N - l_{i1})}{N^2} m_{ir}(1 - m_{ir}) \frac{N^1_{eff} - (N - l_{i1})}{N^1_{eff} - 1}.$$

If $(net_i - m_{i1})/(1 - m_{i1})$ and $net_i/\sqrt{n_0}$ are substituted for m_{ir} and m_{i1}, respectively,

$$\begin{aligned} Var(\hat{net}_i) &= \frac{1}{N}\frac{1-\frac{1}{\sqrt{n_0}}}{1-\frac{net_i}{\sqrt{n_0}}} net_i(1-net_i)\frac{N^1_{eff}-N+l_{i1}}{N^1_{eff}-1} \\ &= \frac{K^1_a K^1_b}{N}\, net_i(1-net_i), \end{aligned} \tag{5.7}$$

where $K^1_a = (1-\frac{1}{\sqrt{n_0}})/(1-\frac{net_i}{\sqrt{n_0}})$ and $K^1_b = (N^1_{eff}-N+l_{i1})/(N^1_{eff}-1)$.

5.2.2 K*th* Hidden Layer

The procedure developed in the previous section for the first hidden layer can be expanded to generalized stochastic models for a synaptic multiplication and a logical OR in the k*th* hidden layer. In the k*th* hidden layer, $E(\hat{m}_{ij})$ and $Var(\hat{m}_{ij})$ can be described by equations 5.2 and 5.3. $Var(\hat{m}_{ij})$ is non-zero in the k*th* hidden layer when $k \geq 2$ because the first term in the right side of equation 5.3 is non-zero due to the random noise introduced by the lower layer. One more assumption is made to take this effect into account, where the input layer and the output layer are regarded as the $0th$ hidden layer and the last hidden layer, respectively.

Assumption 3: The random noise introduced from the lower layer ((k-1)*th* hidden layer) causes α_k to increase such that $\alpha_k \geq \alpha_{k-1}$, while assumptions 1 and 2 are still held. For our analysis, $\alpha_k = \alpha_{k-1} + 0.005$ has been used.

Thus, the effective period of an LFSR in the k*th* hidden layer is

$$\begin{aligned} N^k_{eff} &= N^{k-1}_{eff} + \alpha_k(n_{k-1}-2)N^{k-1}_{eff} \\ &= N^{k-1}_{eff}[1+\alpha_k(n_{k-1}-2)] \\ &= N\prod_{i=1}^{k}[1+\alpha_i(n_{i-1}-2)]. \end{aligned} \tag{5.8}$$

As a result, the generalized forms of equations 5.6 and 5.7 for the kth hidden layer are

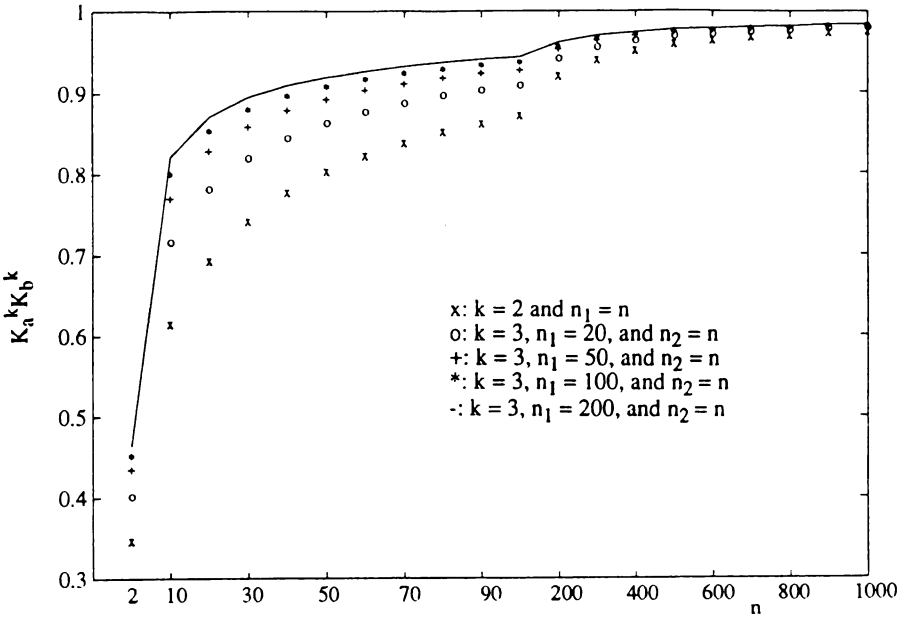
$$E(\hat{net}_i) = 1 - \prod_{j=1}^{n_{k-1}} (1 - m_{ij}) \tag{5.9}$$

and

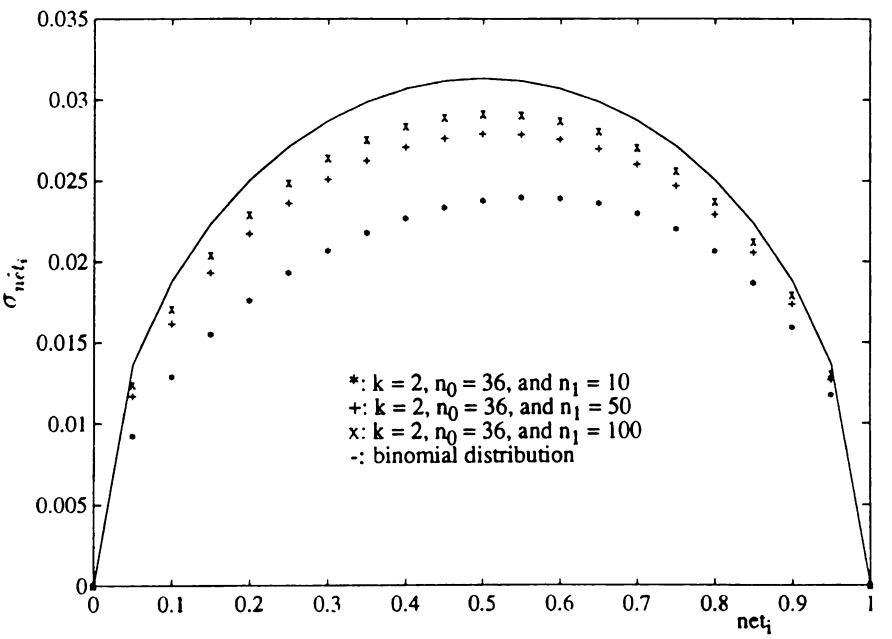
$$Var(\hat{net}_i) = \frac{K_a^k K_b^k}{N} net_i(1 - net_i), \tag{5.10}$$

where $K_a^k = (1 - \frac{1}{\sqrt{n_{k-1}}})/(1 - \frac{net_i}{\sqrt{n_{k-1}}})$ and $K_b^k = (N_{eff}^k - N + l_{i1})/(N_{eff}^k - 1)$.

If statistically independent Bernoulli sequences are assumed, $K_a^k K_b^k = 1$ and the distribution of $\hat{net}_i$ is binomial. In equation 5.10, $Var(\hat{net}_i)$ is bounded to that of the binomial distribution as the n_i's, for $i < k$, become very large. Figure 5.3(a) shows $K_a^k K_b^k$ with respect to n_k and k when $N = 255$, $n_0 = 36$, and $net_i = 0.55$. As seen in Figure 5.3(a), as n_1 and n_2 become large the distribution tends towards the binomial. Figure 5.3(b) shows the standard deviation of $\hat{net}_i$ with respect to net_i for various network configurations. The standard deviation of the binomial distribution is shown for comparison and the standard deviation of $\hat{net}_i$ is skewed to the right because $K_a^k K_b^k$ increases as net_i increases in equation 5.10.



(a)



(b)

Figure 5.3. (a) $K_a^k K_b^k$ with various network configurations when $net_i = 0.55$, $n_0 = 36$, and $k > 1$; (b) standard deviation of $\hat{net}_i$ with respect to net_i.

5.2.3 Neural Activation

Finally, the statistical model of estimation $\hat{v}_i$ of a neuron activation in any layer can be expressed as

$$\begin{aligned} \hat{v}_i &= (net_i^+ + \Delta_{\hat{net_i^+}})\,(1 - net_i^- + \Delta_{\hat{net_i^-}}) \\ &= net_i^+(1 - net_i^-) + net_i^+\Delta_{\hat{net_i^-}} + (1 - net_i^-)\Delta_{\hat{net_i^+}} \end{aligned}$$

where the second-order term is omitted as before. The expected value and variance of $\hat{v}_i$ in the kth hidden layer are

$$E(\hat{v}_i) = net_i^+(1 - net_i^-) \tag{5.11}$$

and

$$Var(\hat{v}_i) = net_i^{+2}Var(\hat{net_i^-}) + (1 - net_i^-)^2 Var(\hat{net_i^+}). \tag{5.12}$$

As n_k and k become very large in equation 5.10, the distributions of $\hat{net_i^+}$ and (1 - $\hat{net_i^-}$) are both approximately binomial. Furthermore, if $P_s = N$ is very large, the distributions are close to Gaussian. Since the random error of $\hat{v}_i$ is approximated as the linear transformation of the random errors of $\hat{net_i^+}$ and $\hat{net_i^-}$ in equation 5.12, the distribution of $\hat{v}_i$ is also close to Gaussian. Figure 5.4(a,b) shows the actual distribution of a neuron in the output layer as obtained by simulation with $v_i = 0.54$ (a) or 0.45 (b), $k = 2, n_0 = 25, n_1 = 5$, and $n_2 = 5$. As Figure 5.4 indicates, the output distribution is observed to be close to Gaussian while the variance is relatively small.

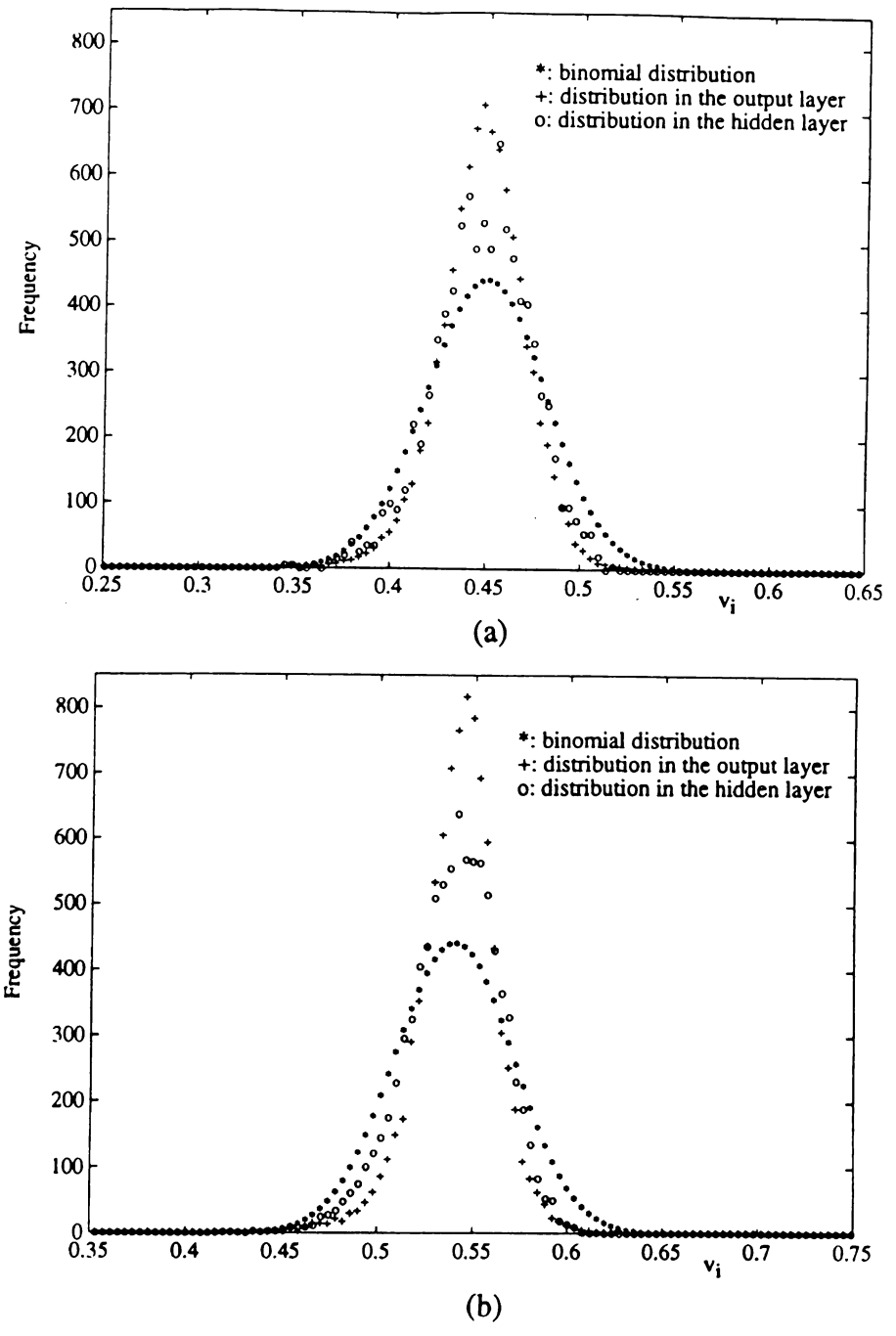


Figure 5.4. The distribution of $\hat{v}_i$ in the hidden layer (o) and the output layer (+) for 8825 tests compared to a binomial distribution when v_i = (a) 0.45 or (b) 0.54, $k = 2$, $n_0 = 25$, $n_1 = 5$, and $n_2 = 5$.

5.3 Network Performance Model

To effectively use this statistical model for analysis, architectural information on the number of layers and the number of neurons in each layer, as well as representational information on the values of the input and target patterns are needed. In addition, some assumptions on the values of net_i^+ and net_i^- must be made. Those values depend on input and target patterns as well as the application problem. In addition to obtaining the variances of estimated outputs in the output layer using equations 5.9 to 5.10, the model will enable the differences between the results obtained from the DMNN and the results obtained from deterministic calculations to be presented.

Binary classification problems are one of the possible target applications of the DMNN and are considered here for analysis. The number of the output neurons corresponds to the number of classifications. 0.45 and 0.55 are used to represent 'on' and 'off' of a neuron in the output layer, respectively. The network is trained in such a way that only one output neuron is 'on' for a given input pattern. After training, the architectural and representational information is fixed. Two network configurations are considered: a two-layer feedforward network with $k = 2, n_0 = 25, n_1 = 5$, and $n_2 = 5$ and a three-layer feedforward network with $k = 3, n_0 = 25, n_1 = 10, n_2 = 5$, and $n_3 = 5$. Tables 5.1 and 5.2 show the standard deviations of the estimated output values in the given network configuration with respect to various possible combinations of net_i^+ and net_i^- leading to the target values (0.45 and 0.55); $\alpha_1 = 0.035$ and $\beta = 1$ are assumed.

Taking the distributions of $\hat{v}_i$('on') and $\hat{v}_i$('off') to be Gaussian, the network correctly classifies the input pattern as the classification i if $\hat{v}_i > 0.5$ and $\hat{v}_j \leq 0.5$ for all $j \neq i$. Consider first the case of a network with two output neurons and define the random variables $V_{on} = \hat{v}_i$('on') and $V_{off} = \hat{v}_i$('off'). The density functions of V_{on}

and V_{off} are

$$f_{V_{on}}(v_{on}) = \frac{1}{\sqrt{2\pi\sigma^2_{V_{on}}}} e^{-(v_{on}-\bar{v}_{on})^2/2\sigma^2_{V_{on}}}$$

and

$$f_{V_{off}}(v_{off}) = \frac{1}{\sqrt{2\pi\sigma^2_{V_{off}}}} e^{-(v_{off}-\bar{v}_{off})^2/2\sigma^2_{V_{off}}},$$

where $\bar{v}_{on}$ and $\bar{v}_{off}$ are the mean values of V_{on} and V_{off} respectively.

Thus the probability of the event $\{V_{on} \leq 0.5\}$ is

$$\begin{aligned} P(V_{on} \leq 0.5) &= F_{V_{on}}(0.5) \\ &= \int_0^{0.5} f_{V_{on}}(\xi) d\xi \end{aligned}$$

and the probability of the event $\{V_{off} > 0.5\}$ is

$$\begin{aligned} P(V_{off} \leq 0.5) &= 1 - F_{V_{off}}(0.5) \\ &= 1 - \int_0^{0.5} f_{V_{off}}(\xi) d\xi, \end{aligned}$$

where $F_{V_{on}}(v_{on})$ is the distribution function of V_{on}. $F_{V_{on}}(v_{on})$ can be described in terms of the standard Gaussian distribution function $F(u)$ by making the variable change

$$u = \frac{(v_{on} - \bar{v}_{on})}{\sigma_{V_{on}}}.$$

Consequently, $F_{V_{on}}(v_{on}) = F(\frac{v_{on}-\bar{v}_{on}}{\sigma_{V_{on}}})$ and $F_{V_{off}}(v_{off}) = F(\frac{v_{off}-\bar{v}_{off}}{\sigma_{V_{off}}})$.

Thus, for a given input pattern, the probability that an input pattern will be correctly classified, P_{cor}, in the network with two output neurons is

$$\begin{aligned} P_{cor} &= P(V_{on} > 0.5 \wedge V_{off} \leq 0.5) \\ &= P(V_{on} > 0.5) P(V_{off} \leq 0.5) \\ &= (1 - F_{V_{on}}(0.5)) F_{V_{off}}(0.5) \end{aligned}$$

$$= (1 - F(\frac{0.5 - 0.55}{\sigma_{V_{on}}}))F(\frac{0.5 - 0.45}{\sigma_{V_{off}}}),$$

where V_{on} and V_{off} are assumed to be statistically uncorrelated.

Therefore, in the general case where the network has n_k output neurons

$$\begin{aligned} P_{cor} &= P(V_{on} > 0.5) \prod_{j \neq i}^{n_k} P(V_{off} \leq 0.5) \\ &= P(V_{on} > 0.5) P(V_{off} \leq 0.5)^{n_k - 1} \\ &= (1 - F_{V_{on}}(0.5)) F_{V_{off}}(0.5)^{n_k - 1} \\ &= (1 - F(\frac{0.5 - 0.55}{\sigma_{V_{on}}}))F(\frac{0.5 - 0.45}{\sigma_{V_{off}}})^{n_k - 1}, \end{aligned} \tag{5.13}$$

where the $\hat{v}_i$'s are assumed to be statistically uncorrelated.

5.4 Simulations

Performance results for the VHDL models developed in Chapter 4 are presented here to demonstrate the validity of the statistical analysis.

Two basic experiments have been conducted. The first involves classifying 5 digits from 0 to 4 while the second involves classifying 10 digits from 0 to 9. The networks are trained with two sets of training data. The first training set consists of ideal digits (1 pattern per digit) and the second consists of the first data set plus 10% noisy patterns (3 patterns per digit). P% noisy patterns were created by randomly inverting P% of the pixels in the ideal digits. For experiment 1, each pattern consists of 6x4 pixels and for experiment 2 each pattern has 7x5 pixels.

Table 5.3 shows P_{cor} for three different network configurations obtained from equation 5.13. In this table, the first column indicates the order of the LFSR, the second column shows P_{cor} directly obtained from equation 5.13 while the third column shows correct classification rates obtained from VHDL simulation. The top row in the table

shows the network configuration in form of n_0 x n_1 x n_2 where n_0, n_1, and n_2 indicate the number of neurons in the input layer, the first hidden layer, and the output layer, respectively. Figures in parenthesis in the third column indicate the correct classification rates when the classification is determined by the neuron with maximum value. Only two-layer networks (one hidden layer and one output layer) are considered here. Each figure in the third column is based on extensive tests (> 20,000 runs). Twenty sets of weights are used and more than 1000 different sets of initial LFSR values per set of weights are used. As shown in Table 5.3, the DMNN character recognizers produce the correct classification rates close to P_{cor}'s in the given network configurations. Differences are mainly caused by the fact that the exact values of parameters α_k and β can not be found although they are bounded for binary classification as described in assumptions 1 and 2. Thus, these values must be selected as arbitrary points within the boundary to calculate N^k_{eff}. To obtain P_{cor} in the table, $\alpha_1 = 0.035, \alpha_2 = 0.04$, and $\beta = 1$ have been used. Thus, the statistical analysis is shown to produce valid results for the DMNN character recognizers.

Table 5.1. Standard deviations of $\hat{v}_i$('on') and $\hat{v}_i$('off') when $n_0 = 25$, $n_1 = 5$, $n_2 = 5$, and (a) $N = 127$; (b) $N = 255$; (c) $N = 511$.

$E(\hat{v}_i) = 0.55$						$E(\hat{v}_i) = 0.45$					
net_i^+	0.6	0.7	0.8	0.9	1.0	net_i^+	0.6	0.7	0.8	0.9	1.0
net_i^-	0.083	0.214	0.313	0.389	0.45	net_i^-	0.25	0.357	0.438	0.5	0.55
$\sigma_{\hat{v}_i}$	0.026	0.025	0.025	0.025	0.026	$\sigma_{\hat{v}_i}$	0.024	0.024	0.025	0.026	0.027
$\sigma_{\hat{v}_i}/\sigma_B^*$	0.587	0.563	0.561	0.572	0.592	$\sigma_{\hat{v}_i}/\sigma_B$	0.543	0.543	0.558	0.581	0.601

(a)

$E(\hat{v}_i) = 0.55$						$E(\hat{v}_i) = 0.45$					
net_i^+	0.6	0.7	0.8	0.9	1.0	net_i^+	0.6	0.7	0.8	0.9	1.0
net_i^-	0.083	0.214	0.313	0.389	0.45	net_i^-	0.25	0.357	0.438	0.5	0.55
$\sigma_{\hat{v}_i}$	0.018	0.018	0.018	0.018	0.018	$\sigma_{\hat{v}_i}$	0.017	0.017	0.018	0.018	0.019
$\sigma_{\hat{v}_i}/\sigma_B$	0.587	0.563	0.560	0.572	0.592	$\sigma_{\hat{v}_i}/\sigma_B$	0.543	0.543	0.557	0.581	0.609

(b)

$E(\hat{v}_i) = 0.55$						$E(\hat{v}_i) = 0.45$					
net_i^+	0.6	0.7	0.8	0.9	1.0	net_i^+	0.6	0.7	0.8	0.9	1.0
net_i^-	0.083	0.214	0.313	0.389	0.45	net_i^-	0.25	0.357	0.438	0.5	0.55
$\sigma_{\hat{v}_i}$	0.013	0.012	0.012	0.013	0.013	$\sigma_{\hat{v}_i}$	0.012	0.012	0.012	0.013	0.013
$\sigma_{\hat{v}_i}/\sigma_B$	0.586	0.562	0.560	0.572	0.592	$\sigma_{\hat{v}_i}/\sigma_B$	0.542	0.542	0.557	0.581	0.609

(c)

* σ_B indicates the standard deviation of $\hat{v}_i$ when Bernoulli sequences are used.

Table 5.2. Standard deviations of $\hat{v}_i$('on') and $\hat{v}_i$('off') when $n_0 = 25, n_1 = 10, n_2 = 5$, $n_3 = 5$, and (a) $N = 127$; (b) $N = 255$; (c) $N = 511$.

$E(\hat{v}_i) = 0.55$						$E(\hat{v}_i) = 0.45$					
net_i^+	0.6	0.7	0.8	0.9	1.0	net_i^+	0.6	0.7	0.8	0.9	1.0
net_i^-	0.083	0.214	0.313	0.389	0.45	net_i^-	0.25	0.357	0.438	0.5	0.55
$\sigma_{\hat{v}_i}$	0.029	0.028	0.028	0.029	0.030	$\sigma_{\hat{v}_i}$	0.027	0.027	0.028	0.029	0.031
$\sigma_{\hat{v}_i}/\sigma_B^*$	0.668	0.640	0.639	0.655	0.681	$\sigma_{\hat{v}_i}/\sigma_B$	0.621	0.619	0.635	0.661	0.694

(a)

$E(\hat{v}_i) = 0.55$						$E(\hat{v}_i) = 0.45$					
net_i^+	0.6	0.7	0.8	0.9	1.0	net_i^+	0.6	0.7	0.8	0.9	1.0
net_i^-	0.083	0.214	0.313	0.389	0.45	net_i^-	0.25	0.357	0.438	0.5	0.55
$\sigma_{\hat{v}_i}$	0.020	0.020	0.020	0.020	0.021	$\sigma_{\hat{v}_i}$	0.019	0.019	0.020	0.021	0.022
$\sigma_{\hat{v}_i}/\sigma_B$	0.667	0.640	0.638	0.654	0.681	$\sigma_{\hat{v}_i}/\sigma_B$	0.621	0.619	0.634	0.661	0.694

(b)

$E(\hat{v}_i) = 0.55$						$E(\hat{v}_i) = 0.45$					
net_i^+	0.6	0.7	0.8	0.9	1.0	net_i^+	0.6	0.7	0.8	0.9	1.0
net_i^-	0.083	0.214	0.313	0.389	0.45	net_i^-	0.25	0.357	0.438	0.5	0.55
$\sigma_{\hat{v}_i}$	0.015	0.014	0.014	0.014	0.015	$\sigma_{\hat{v}_i}$	0.014	0.014	0.014	0.014	0.015
$\sigma_{\hat{v}_i}/\sigma_B$	0.667	0.640	0.638	0.654	0.680	$\sigma_{\hat{v}_i}/\sigma_B$	0.620	0.618	0.634	0.660	0.693

(c)

* σ_B indicates the standard deviation of $\hat{v}_i$ when Bernoulli sequences are used.

Table 5.3. P_{cor} obtained from equation 5.13 and correct classification rates from VHDL simulations.

25 x 5 x 5 for 5-digit classification		
	P_{cor} obtained from equation 5.13	correct classification rate obtained from VHDL simulations
8-bit	86.55% ~ 90.91%	92.5%(97.9%)
9-bit	98.59% ~ 99.18%	99.3%(99.65%)

36 x 9 x 10 for 10-digit classification		
	P_{cor} obtained from equation 5.13	correct classification rate obtained from VHDL simulations
8-bit	86.64% ~ 92.94%	86.73%(97.8%)
9-bit	97.93% ~ 99.74%	92.6%(99.94%)
10-bit	100%	98.7%(99.06%)

36 x 30 x 10 for 10-digit classification		
	P_{cor} obtained from equation 5.13	correct classification rate obtained from VHDL simulations
8-bit	74.25% ~ 83.7%	80.27%(92.4%)
9-bit	96.39% ~ 98.64%	91.2%(99.1%)
10-bit	99.91% ~ 100%	99.1%(99.6%)

CHAPTER 6

DMNN Application: Pattern Classification

A DMNN architecture can be viewed as a coprocessor with the addition of a control unit and memory components. A DMNN coprocessor for classifying binary patterns is modeled and simulated in VHDL. A general design procedure for constructing DMNN binary classifiers is discussed. XOR and encoding problems are applied as testbench problems. Then numeric character classification problems are applied. Convergence properties in DMNNs during training are also discussed. Classification performance for these networks is evaluated in comparison to that obtained from deterministic DMNN simulations and ordinary back-propagation networks.

6.1 Introduction

The purpose of pattern classification is to perform a mapping from observation space to interpretation space by extracting features from observed data and classifying the collected features into a certain category [85]. An important aspect of pattern

classification is the determination of the discriminant (or operator) that produces an estimate of the class membership of an input pattern. Some of the early work in pattern classification was inspired by the idea that a network of processing elements arranged in a manner similar to a biological neural network might be able to learn the requisite operators in an autonomous manner.

The Perceptron was the first significant development of such in the early 1960s, suggesting a general approach to the automatic learning discriminants for pattern classification [42]. However, at that time, even though the power of the generalized Perceptron devices were well understood, no practical method for training layered Perceptron devices was known. Furthermore, the analysis of single-layer Perceptron devices by Minsky and Papert in 1969 revealed the limitation of the Perceptron [2]. That is, a single-layer Perceptron can not classify linearly unseparable patterns. Since then, no significant further progress in Perceptron-related research had been achieved until the early 1980s.

The early disappointments with the Perceptron approach caused most traditional pattern recognition researchers to concentrate on geometric or structural approaches. In these approaches, pattern recognition systems made use of the results of statistical communication and estimation theory or mathematical linguistics.

In 1986, Rumelhart, *et al.* introduced a new algorithm, called the generalized delta rule (or back-propagation), for training multilayer Perceptrons. The means of autonomous mapping from observation space to interpretation space is now possible. However, learning using the ordinary back-propagation algorithm is impractically slow. Many versions of the ordinary back-propagation have been developed to accelerate learning [43].

One current research emphasis is on implementing pattern classification systems using the neural network approach on advanced computer architectures or in dedicated VLSI hardware. In this chapter, binary classification problems are applied to

the DMNN to test its applicability to pattern classification using the VHDL model of the DMNN described in previous chapters.

6.2 Methodology

A generic DMNN architecture has been modeled as a coprocessor in VHDL. Binary classification problems are applied to the DMNN coprocessor as testbench problems. In order to construct the network for solving a particular classification problem, sampling data (or patterns) must be selected carefully. The sampling data are often divided into training data and testing data. The DMNN must be trained off-line on a host computer. Once training is completed, test data are applied to the DMNN coprocessor whose network architecture is set up as a result of training. Based on the test results, the performance of a DMNN pattern classifier is evaluated.

A back-propagation algorithm has been programmed in C. Each pattern is of an array of binary values. The number of neurons in the input layer of a DMNN equals the number of elements in the pattern vector plus one. The additional input is required to provide the synaptic elements located in upper layers with a constant threshold input (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. Only one output neuron is turned on when an input pattern is applied to the network. For example, if an input pattern belongs to category # 1, only the first output neuron is turned on and all the other neurons are turned off. The values representing 'on' and 'off' of output neurons are determined during the training session. These values can be any intermediate values between 0.0 and 1.0.

6.2.1 Training and Classification

A DMNN must be trained with training and target patterns before it is used for classification. During training, network configuration, synaptic weights, and values representing the 'on' and 'off' thresholds of output neurons are determined. This information is needed to construct the DMNN for testing or classifying patterns. Training requires the following steps:

Step 1. Start with an initial network configuration.

Step 2. Initialize synaptic weights to random values generated between 0.0 and 1.0.

Step 3. Apply training patterns and corresponding target patterns one by one.

Step 4. Once all the training patterns are applied, perform one pass of feedforward operations and calculate a sum-squared error of the network. Then propagate the error in a backward direction and modify the synaptic weights.

Step 5. Repeat step 4 iteratively until the network converges in such a way that the sum-squared error reaches a predefined error-threshold. If the network oscillates or is stuck at a high local minimum in error surface, increase the number of neurons in the hidden layers (and the number of hidden layers if necessary)* and repeat steps 2 to 5.

Step 6. Once the network reaches the predefined error-threshold, check to see if all the synaptic weights are valid, i.e., $-1.0 \leq w_{ij} \leq 1.0$ for all i and j. If one or more synaptic weights are not valid, increase the number of neurons in the

*See reference [43] for detailed procedures suggested by some researchers.

hidden layers (and the number of hidden layers if necessary) and repeat steps 2 to 6.

Step 7. Save the final synaptic weights and the network configuration.

Our experimental experience suggests two layers (one hidden layer and one output layer) are enough for binary classification using the DMNN architecture. It has also been observed, through extensive training with an arbitrary size DMNN, that there is a relationship between the existence of solutions, equivalently low sum-squared error states, and the ratio of an input range and an output range. 0.1 was found to work well as the ratio as this always produced valid trained weights with the given input pattern representations. 0.0 and 1.0 have been used to represent '0' and '1' binary values respectively. 0.45 and 0.55 have been used as target values corresponding to 'on' and 'off' thresholds at the output neurons.

Once the training has been completed, the network configuration and synaptic weights obtained from the training are loaded into the DMNN coprocessor. Then test patterns are applied to the network to see how well the network classifies them.

Define the minimal network configuration for solving a particular problem as the network that converges to an error state equal to or below the predefined error threshold with a mimum number of neurons in the hidden layers and a minimum number of hidden layers. A network configuration can be represented by n_0 x n_1 x $\cdots$, where n_0 and n_1 denote the number of elements in an input pattern and the number of neurons in the first hidden layer, respectively.

In following two sections, some benchmark problems and character classification problems are applied and their performance is evaluated; only minimal network configurations for these problems are illustrated.

6.3 Benchmark Problems

Two benchmark problems are applied to the DMNN coprocessor to test its ability classifying linearly unseparable patterns (XOR problem) and to test whether or not the DMNN can work as a data compressor (Encoding problem). In these testbench problems, the training data and the testing data are the same.

6.3.1 DMNN XOR Problem Solver

The XOR problem is a typical example of 4 input patterns that can not be correctly classified by a single-layer Perceptron. Thus, it is often used to test whether or not an artificial neural network can classify linearly unseparable data. Table 6.1 shows the representation of the XOR problem in a DMNN. Figure 6.1 shows an example of the two-layer DMNN for solving the XOR problem, where the values of synaptic weights are set at one before quantization is applied.

Table 6.2 shows the actual outputs of the network illustrated in Figure 6.1 with respect to the register length n. In Table 6.2, each figure is rounded to 3 decimal places. As shown in Table 6.2, when $n \geq 7$, the DMNN can classify the 4 input patterns correctly. However, when $n \leq 6$, the network misclassifies one of them be-

Table 6.1. Representation of the XOR problem in a DMNN.

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

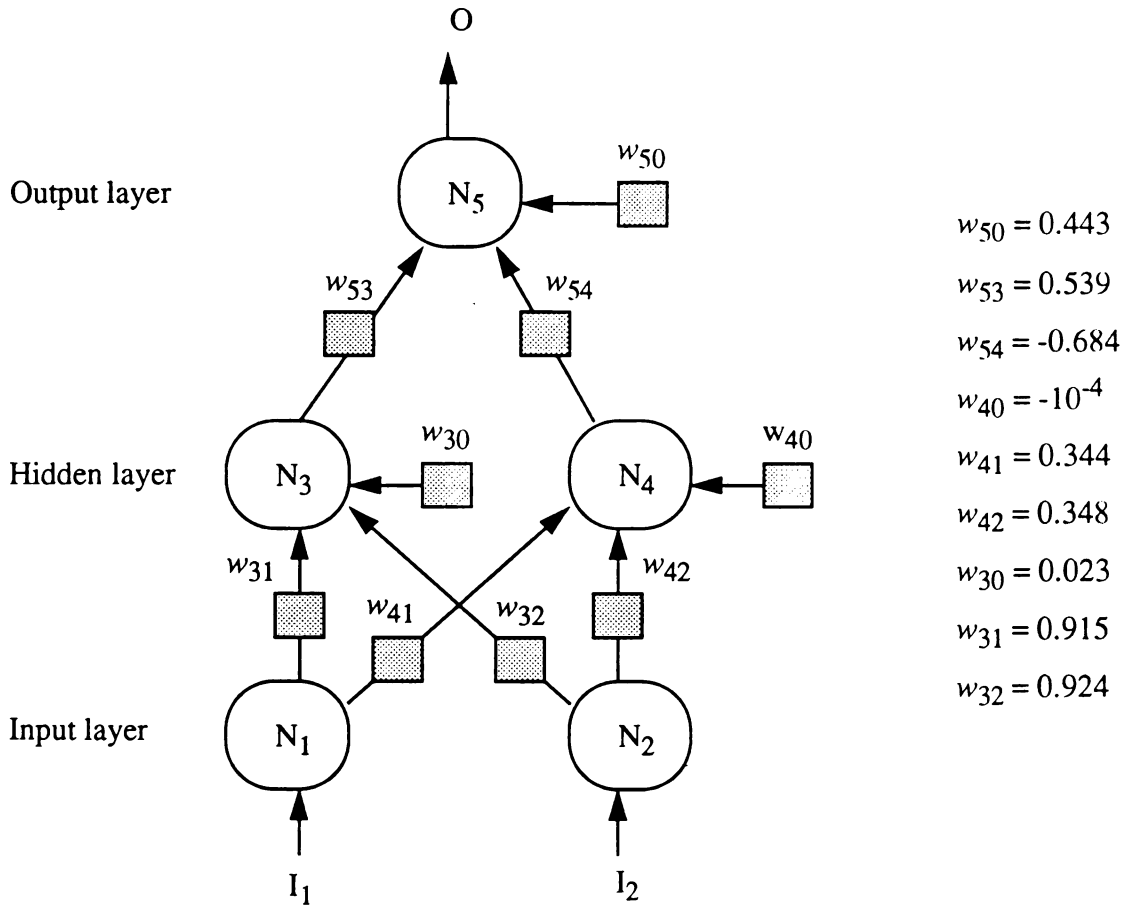


Figure 6.1. An example DMNN for solving the XOR problem.

Table 6.2. Actual outputs of an n-bit two-layer DMNN for solving XOR problem.

		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

cause errors on synaptic weights caused by quantization become too large. The minimal network configuration for solving the XOR problem is 3 x 2 x 1.

6.3.2 DMNN Encoder

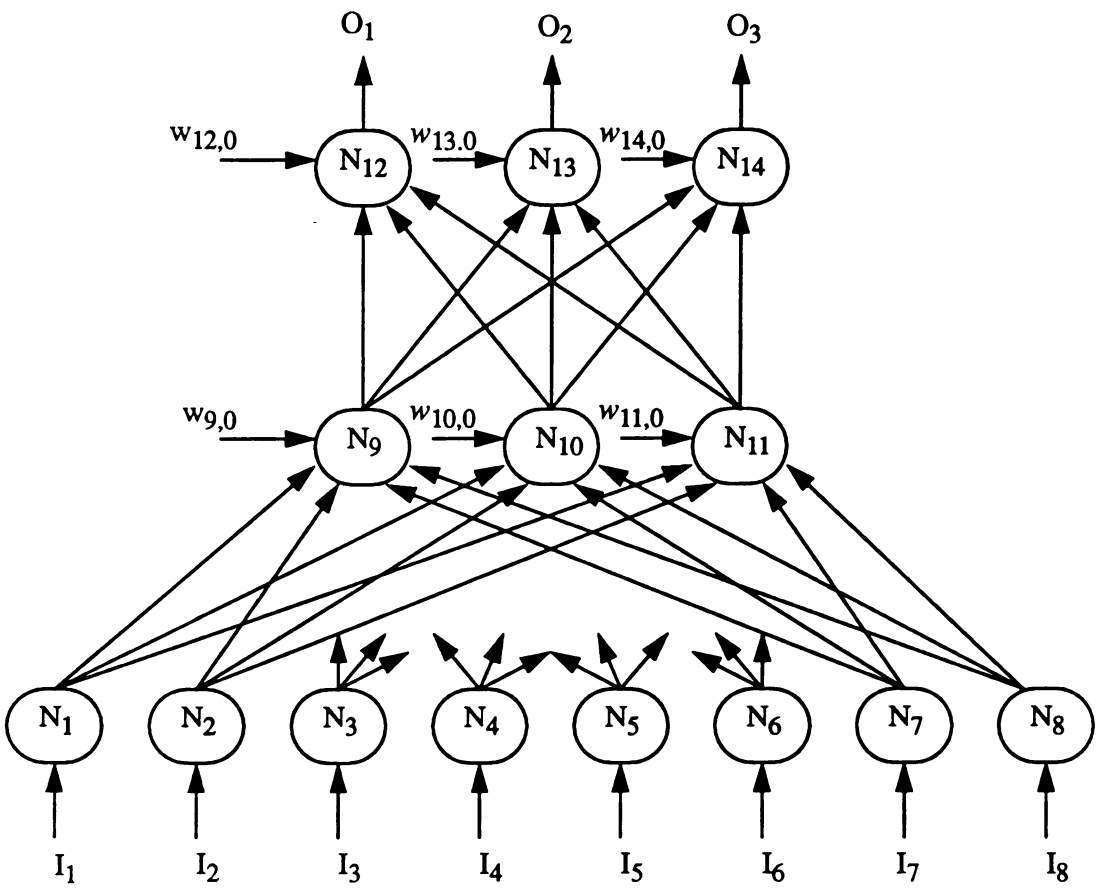
An encoding problem is used here to test whether or not the DMNN has the ability to compress data. An 8-to-3 encoding problem is considered here as an example. Table 6.3 shows the representation of the problem in the DMNN.

Figure 6.2 shows an example of the two-layer DMNN for solving the 8-to-3 encoding problem. The minimal network configuration is 9 x 3 x 3. The DMNN successfully encodes 8-bit data to 3-bit data when $n \geq 8$, verifying that the DMNN is able to compress data.

Data for network configurations, training and test patterns, and initial LFSR values for solving the XOR and the 8-to-3 encoding problems are listed in Appendix D.

Table 6.3. Representation of the 8-to-3 encoding problem in a DMNN.

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



$w_{9,1} = -0.0026$ $w_{10,1} = -0025$ $w_{11,1} = -0.0442$ $w_{12,9} = -0.0413$ $w_{14,9} = 0.0001$
$w_{9,2} = -0.0018$ $w_{10,2} = 0.3106$ $w_{11,2} = -0.0201$ $w_{12,10} = 0.0207$ $w_{14,10} = 0.5928$
$w_{9,3} = 0.3277$ $w_{10,3} = -0.0016$ $w_{11,3} = 0.0149$ $w_{12,11} = 0.6153$ $w_{14,11} = 0.0164$
$w_{9,4} = 0.3240$ $w_{10,4} = 0.3107$ $w_{11,4} = -0.0001$ $w_{12,0} = 0.4310$ $w_{14,0} = 0.4481$
$w_{9,5} = -0.0006$ $w_{10,5} = -0.0003$ $w_{11,5} = 0.2971$
$w_{9,6} = -0.0011$ $w_{10,6} = 0.3032$ $w_{11,6} = 0.2867$ $w_{13,9} = 0.5892$
$w_{9,7} = 0.2861$ $w_{10,7} = -0.0007$ $w_{11,7} = 0.3223$ $w_{13,10} = 0.0101$
$w_{9,8} = 0.2822$ $w_{10,8} = 0.3022$ $w_{11,8} = 0.3129$ $w_{13,11} = 0.0811$
$w_{9,0} = 0.0001$ $w_{10,0} = 0.0001$ $w_{11,0} = 0.0563$ $w_{13,0} = 0.4406$

Figure 6.2 An example DMNN for solving the 8-to-3 encoding problem.

6.4 DMNN Character Recognizer

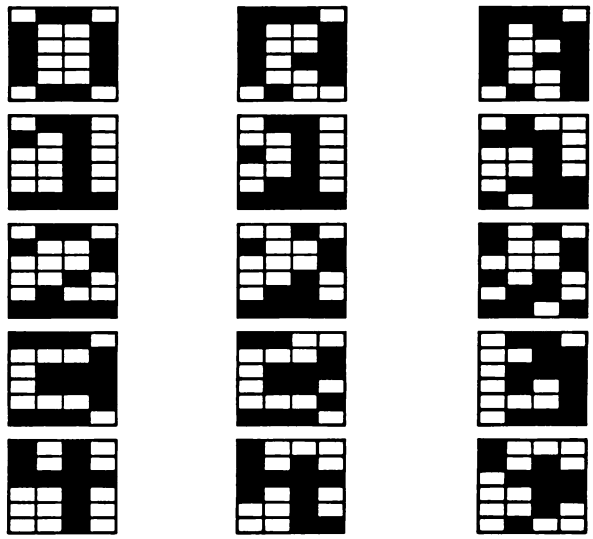
Two experiments have been conducted involving the classification of numeric characters. The first experiment involves classifying 5 digits from 0 to 4 while the second experiment involves classifying 10 digits from 0 to 9.

6.4.1 Data Set

For each experiment, a network is trained with two sets of training data. The first set consists of ideal digits (1 pattern per digit) and the second consists of the first data set plus 10% noisy patterns (3 patterns per digit). The test data set is composed of the first data set, the second data set, and 20 % noisy patterns (3 patterns per digit). P% noisy patterns are created by randomly inverting P% of the pixels in the ideal digits. For experiment 1, each pattern consists of 6x4 pixels and for experiment 2 each pattern has 7x5 pixels. Figure 6.3 and 6.4 show the pixel images of a typical data set used in experiments 1 and 2, respectively. The average Hamming distances between ideal digits are 10.2 and 12.1 for experiment 1 and 2, respectively.

6.4.2 Experimental Results and Network Performance

Experimental results are compared to results from the program 'annclass' simulating ordinary multilayer neural networks built on the Rochester Connectionist Simulator. A continuous sigmoid function is used as a neuron transfer function in the comparison simulation. The purpose of experiment 1 is to find the relationship between network convergence and network configuration.

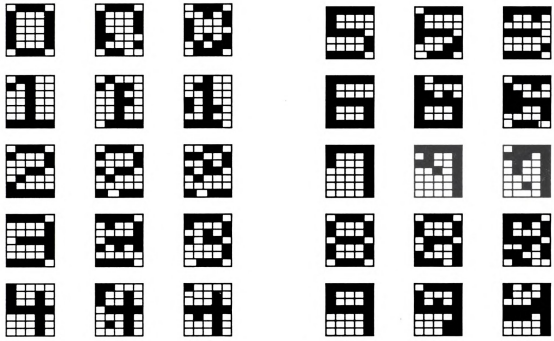


(a) (b) (c)

digit	"0"	"1"	"2"	"3"	"4"
"0"	0	13	9	10	13
"1"	13	0	6	11	8
"2"	9	6	0	9	12
"3"	13 10	11	9	0	11
"4"		8	12	11	0

(d)

Figure 6.3. For 5-digit classification in experiment 1: (a) ideal digits; (b) typical 10% noisy digits; (c) typical 20 % noisy digits; (d) Hamming distance between ideal digits.



(a) (b) (c) (a) (b) (c)

digit	"0"	"1"	"2"	"3"	"4"	"5"	"6"	"7"	"8"	"9"
"0"	0	17	11	10	19	11	9	9	5	11
"1"	17	0	10	15	20	14	16	18	16	20
"2"	11	10	0	9	20	12	12	14	8	14
"3"	10	15	9	0	17	7	9	11	7	9
"4"	19	20	20	17	0	14	16	18	18	14
"5"	11	14	12	7	14	0	4	12	8	8
"6"	9	16	12	9	16	4	0	12	8	8
"7"	9	18	14	11	18	12	12	0	12	4
"8"	5	16	8	7	18	8	8	12	0	10
"9"	11	20	14	9	14	8	8	4	10	0

(d)

Figure 6.4. For 10-digit classification in experiment 2: (a) ideal digits; (b) typical 10% noisy digits; (c) typical 20 % noisy digits; (d) Hamming distance between ideal digits.

Table 6.4. Average number of iterations required for training in experiment 1.

DMNN						'annclass'		
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

Table 6.4 shows the average number of iterations required to converge with a given network configuration for the DMNN and 'annclass' when they are trained with the 5 ideal digits. Each figure in the table is the average value based on 10 different initial sets of weights. For the DMNN, η (learning rate) = 0.2, α (momentum parameter) = 0.5, and *Err_td* (error threshold) = 10^{-4} are used, while $\eta = 1.0$, $\alpha = 0.5$, and *Err_td* = 10^{-2} are used for 'annclass'. The use of smaller values of η and *Err_td* for training the DMNN is affected by the constraints in its operation range. In Table 6.4, * indicates a minimal network configuration, and N_1 and N_2 are the numbers of neurons in the first and second hidden layers, respectively.

As shown in Table 6.4, the number of iterations increases in general as the number of hidden layers increases or the number of neurons in the hidden layer decreases in DMNNs, but there is no clear relationship between the network configuration and the number of iterations in 'annclass'. It is noted that the DMNN network can be train-

Table 6.5. Performance of the DMNN 5-digit recognizer.

Trained with the ideal digits			
	Resubstitution error rate	Misclassified error rate on 10% noisy data	Misclassified error rate on 20% noisy data
8-bit DMNN	7.5%	32.2%	55.6%
9-bit DMNN	0.7%	11.7%	30%
C simulation	0%	8.5%	43%
Trained with the ideal digits plus 10% noisy data			
	Resubstitution error rate		Misclassified error rate on 20% noisy data
8-bit DMNN	5%		38.7%
9-bit DMNN	0%		26.7%
C simulation	0%		35.3%

ed with a significantly reduced number of iterations compared to 'annclass', which requires approximately 14000 (average) iterations for two-layer networks and 55000 for three-layer networks. The fast convergence obtained in the DMNN results from the fact that the values of output neurons can take on intermediate values between 0 and 1 as targets.

Table 6.5 shows the performance results of the DMNN 5-digit recognizer with 8 and 9 bit register lengths. The results of a deterministic DMNN simulation (in C) directly calculating the network operations are also given for comparison. The classification of test patterns is determined by the MSBs of the magnitude parts of the numbers in the output neurons. If the output of a neuron in the output layer is greater than or equal to 0.5, the neuron is considered to be 'on'. If no neuron or more than

1 neuron are turned 'on', the test is considered a misclassification. Typical synaptic weights and network configuration for 5-digit classification are listed in Appendix D.

Each figure in Table 6.5 is based on 20 tests. For each test, new synaptic weights are loaded. Resubstitution error is obtained by testing the DMNN with the training data set. As shown in this table, the DMNN character recognizer with a 9-bit register length produces results whose performance level is close to that of the deterministic simulation on resubstitution data. It is noted that the performance of the 9-bit DMNN is better than that of the deterministic simulation on 20% noisy data. The reason is perhaps that the random noise may help enhance the network performance. It is also shown that the DMNN classifies patterns better when the training data contain noisy data. Meanwhile, when the register length is less than 8 bits, the misclassification error rate is more than 50%, even on resubstitution tests.

For experiment 2, the 5-digit classification problem is extended to a 10-digit classification problem and the pixel images of each pattern are increased to 7x5. The network performance is again compared with that of the deterministic DMNN simulation and 'annclass' in terms of a correct classification rate. Table 6.6 shows the performance results of the DMNN 10-digit recognizer when the register length is 8, 9, or 10. For training, $\eta = 0.05$, $\alpha = 0.0$, and $Err\_td = 0.005$ are used. Only two-layer networks (one hidden layer and one output layer) are considered here. For the DMNN, the minimal network configurations are 36 x 9 x 10 and 36 x 30 x 10 when it is trained with the first set of training data and the second set of training data, respectively. The number of neurons in the hidden layer for 'annclass' ranges from 5 to 20 for both. The first three rows show the results obtained from the DMNN coprocessor, followed by the results of the deterministic simulation and data from 'annclass'. An example of synaptic weights and network configuration for 10-digit classification is listed in Appendix D.

Table 6.6. Performance of the DMNN 10-digit recognizer.

Trained with the ideal digits on the 36x9x10 DMNN			
	Resubstitution error rate	Misclassified error rate on 10% noisy data	Misclassified error rate on 20% noisy data
8-bit DMNN	2.2%	16.2%	43.3%
9-bit DMNN	0.06%	9.7%	36.6%
10-bit DMNN	0.94%	7.8%	37.1%
C simulation	0.0%	6.7%	35.0%
annclass	0.0%	15.8%	30.8%

Trained with the ideal digits plus 10% noisy data on the 36x30x10 DMNN		
	Resubstitution error rate	Misclassified error rate on 20% noisy data
8-bit DMNN	7.6%	25.4%
9-bit DMNN	0.9%	20.3%
10-bit DMNN	0.4%	15.1%
C simulation	0.0%	13.5%
annclass	0.0%	16.7%

Each entry in Table 6.6 is based on extensive tests (> 20,000 runs). Twenty sets of weights are used and more than 1000 different sets of initial LFSR values per set of weights are used. Correct classification is determined by the output neuron with a maximum value and the test is considered a misclassification when an incorrect neuron has a maximum value. As shown in Table 6.6, the DMNN character recognizer with a 9-bit register length produces results whose performance level is close to that of the deterministic simulation and competitive to that of the 'annclass'. The correct classification rates obtained from the DMNN are close to those for the deterministic

simulation, which indicates the quantization does not have significant effect on the network performance when the register length is greater than 8. Better performance from the DMNN than average values shown in Table 6.6 can be obtained by choosing a set of initial values of LFSR's carefully for a given set of weights.

6.5 Summary

Digital multilyer neural networks for binary classification problems have been modeled and simulated in VHDL. Any size feedforward neural network can be constructed by simply interconnecting the desired number of predefined modules. All operations in the same layer are performed in parallel and all operations between the layers are performed in a pipelined manner. Thus, the DMNN architecture utilizes full parallelism embedded in ANN computations. Its processing speed depends only on the clock frequency and the register length, not on the network size. When an n-bit register length and W MHz clock-cycle are used for the DMNN character recognizer, the processing speed is about $\frac{W \text{x} 10^6}{2^n - 1}$ patterns per second.

The DMNN architecture is also extremely compact. The chip areas required by 10-digit DMNNs are less than 1.0 x $10^5 A_g$ and 3.0 x $10^5 A_g$ for 36 x 9 x 10 and 36 x 30 x 10 network configurations, respectively, as was determined in section 4.5. The classification performance of two-layer DMNNs for 5-digit and 10-digit classifications is found to be competitive to that of deterministic DMNN simulations and ordinary back-propagation networks when the register length is greater than 7.

CHAPTER 7

Conclusion

A new architecture for a VLSI-based digital multilayer neural network has been developed in this dissertation. Statistical models for stochastic computations utilized in this architecture have been developed. Network analysis has been performed based on the statistical models. VHDL models of DMNN pattern classifiers indicate that the classification performance of this architecture is competitive to that of deterministic simulations or ordinary back-propagation networks while retaining the desirable properties of high speed and high density on a chip. This chapter begins with a summary of the research work, followed by an identification of the major contributions of this work. A discussion of future research issues concludes the chapter.

7.1 Summary

The development of fast, space-efficient, and programmable architectures for dedicated VLSI ANNs which are applicable to real engineering problems is a current challenge in ANN research. Analog VLSI ANNs have the potential for high speed and high density on a chip. However, the unavailability of reliable permanent analog

storage devices makes it difficult to build programmable analog ANNs and design parameter variations, such as noise and temperature, diminish the accuracy in analog computations. Furthermore, high parasitic capacitances on external I/O pins and difficulties in designing large analog systems make it difficult to build large size or multi-chip ANNs. A conventional digital approach using ALU units or special hardware as computing elements leads to increased hardware requirements and decreased parallelism. A promising solution to these problems has been demonstrated in this research work.

A pulse-mode digital multilayer neural network (DMNN) architecture, which can be implemented using simple logic gates and simplistic digital components, has been developed. The DMNN performs pseudo-analog computations using stochastic computing techniques and random pulse streams. The modular DMNN architecture is programmable and compact in size. Any size feedforward neural network can be built by interconnecting the desired number of modules.

In the DMNN, all operations in the same layer are performed in parallel and all operations between the layers are performed in pipeline. Thus full parallelism is utilized. Processing speed depends only on the clock frequency and register length, not on the network size. When the network is used for pattern classification, about 200,000 patterns per second can be classified with a 50 MHz clock assuming an 8-bit register length.

Statistical models of operations occurring in the DMNN have also been developed. The variance of the random noise (or error) resulting from the estimation of the generating probabilities has been shown to be bounded to the binomial distribution which can be obtained when Bernoulli sequences are assumed. Network analysis for binary classification has been performed. Experimental results show that the successful recovery rate of target patterns for training data closely matches what was anticipated by the analytical results. It is also shown that the low variances in estimated compu-

tational results in the DMNN lead to the high success rate in the recovery of target patterns.

The DMNN has been applied successfully to binary classification problems such as XOR, encoding, and character classification. Two-layer DMNNs for these problems have been simulated in VHDL. Their performance has been found to be competitive to that obtained from deterministic DMNN simulations and ordinary back-propagation networks when the register length is greater than 7.

The hardware requirement for implementing DMNNs is significantly reduced compared to other digital approaches. For example, the 8, 9, and 10-bit DMNNs, whose network configuration is 36 x 30 x10, can be built using about 2.27 x 10^5, 2.53 x 10^5, and 2.80 x 10^5 NAND gates, respectively. This indicates that a 10-bit register length DMNN for 10-digit recognition can be built on a single chip.

7.2 Contributions

The major contributions of this research are:

1. A VLSI-based pulse-mode digital multilayer neural network (DMNN) architecture has been developed. Stochastic computing techniques and use of simple logic gates as computing elements lead to a space-efficient network architecture. The DMNN is programmable so that many network configurations can be formed and its modular architecture makes the networks expandable to any size.

2. Statistical models for estimated probabilities resulting from network operations have been developed using the statistical dependence of pulse occurrence in a pseudo-random pulse stream generated from a tapped LFSR and the determin-

istic nature observed in gate operations. The random noise effects on network operations have been quantified.

3. VHDL has been used as a modeling tool for a DMNN coprocessor. The hierarchical design capability supported by VHDL, combined with the natural modularity of the DMNN architecture makes design analysis simple.

4. Character classification problems have been applied to this model and it has been shown that the classification performance of DMNN 5-digit and 10-digit classifiers is competitive to that obtained from deterministic DMNN simulations and ordinary back-propagation networks while retaining the desired high speed and high hardware density.

7.3 Future Research

The DMNN architecture has been developed in a mixed mode using gate-level and register-transfer level designs, but it can be fully described at the gate-level. To implement the DMNN architecture as a dedicated VLSI ANN, two approaches are possible: custom-design and semicustom-design. The DMNN with a minimal network configuration, dedicated to a particular application problem, can be built on a single chip using a custom-design methodology. But, a semicustom-design method is more appropriate when modularity or expandability is a more important factor than size. Any size network can be built using the defined modules. These modules can be implemented using gate arrays and/or standard cells.

There is no reason that feedback networks, such as the Hopfield network or the Boltzmann machine, can not be constructed using the modules developed in this work. However, in order for the modules to be used for feedback networks, statistical

models for analizing the random noise effects caused by stochastic computations on network dynamics must be developed.

Finally, DMNN binary classifiers described in this thesis may be further used for solving more complex pattern classification problems. In an n-bit DMNN, $2^n - 1$ levels are available to represent the values of elements in input and output patterns. Thus, each multi-level input pattern can contain more information or can represent a more complex pattern. Furthermore, multi-level output neurons can make it possible to represent multiple categories in an encoded form using fewer output neurons. A simple example of such was illustrated in the 8-to-3 encoder in Chapter 6. However, in order to make use of this advantage, one condition must be satisfied. That is, networks with given input and output data representations must converge during training to an equilibrium point in the error surface which is equal to or below a desired error threshold.

APPENDICES

APPENDIX A

Derivation of the rth Factorial Moment of a Hypergeometric Random Variable X

The probability function for a hypergeometric random variable X is

$$P(X = k) = \frac{\binom{l}{k}\binom{N-l}{n-k}}{\binom{N}{n}}$$

where l is a natural number (*i.e.*, $l \in 0, 1, 2, \ldots, N-1, N$).

Let $(k)_r$ be the product of r consecutive integers starting with k such that $(k)_r = k(k-1)\cdots(k-r+1)$. Then, the rth factorial moment is

$$\begin{aligned} E[(X)_r] &= \sum_{k=r}^{N} (k)_r P(X = l) \\ &= \sum_{k=r}^{N} (k)_r \frac{\binom{l}{k}\binom{N-l}{n-k}}{\binom{N}{n}}. \end{aligned} \tag{A.1}$$

By using the identity

$$(k)_r\binom{l}{k} = (l)_r\binom{l-r}{k-r},$$

we can obtain

$$\begin{aligned}
\sum_{k=r}^{N}(k)_r\binom{l}{k}\binom{N-l}{n-k} &= \sum_{k=r}^{N}(l)_r\binom{l-r}{k-r}\binom{N-l}{n-k} \\
&= (l)_r\sum_{j=0}^{N}\binom{l-r}{j}\binom{(N-r)-(l-r)}{(n-r)-j} \\
&= (l)_r\binom{N-r}{n-r}\sum_{j=0}^{N}\frac{\binom{l-r}{j}\binom{(N-r)-(l-r)}{(n-r)-j}}{\binom{N-r}{n-r}} \\
&= (l)_r\binom{N-r}{n-r} \\
&= \frac{(l)_r(n)_r\binom{N}{n}}{(N)_r}. \qquad \text{(A.2)}
\end{aligned}$$

Thus, substituting equation A.2 for equation A.1 gives

$$E[(X)_r] = \frac{(l)_r(n)_r}{(N)_r}.$$

APPENDIX B

Program for DMNN Back-Propagation Training

```
/*  This program, named DMNN, simulates the back-propagation
    learning algorithm training and tesing digital multilayer neural
    Networks.  Momentum term is added for weight changes.

     Command line:

       DMNN net_con input_pattern target_pattern weight_file,
        where net_con indicates the network configuration.
*/

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#define features 24
#define pattern_no 40
#define neuron_no 56
#define layer_no 2
#define in_layer 35
#define h1_layer 10
#define h2_layer 0
#define out_layer 10
#define RAND_MAX 32767

            /* External variable declaration */

int con_flag=0,net_con[neuron_no][neuron_no],disp_no, seed;
float inpattern[pattern_no][neuron_no],weights[neuron_no]
      [neuron_no];
float target[pattern_no][neuron_no],eta,gamma,tor_error;
float net_ex[pattern_no][neuron_no],net_in[pattern_no][neuron_no];
float potential[pattern_no][neuron_no],new_weights[neuron_no]
      [neuron_no];
FILE *fp,*fp2;

            /* Main Program */

main(argc,argv)
int argc;
char *argv[];

 {  int i,j,k,cycle,invalid_wgts;
    char learn_flag, test_yes;

 if(argc != 4  & & argc != 5) {
    fprintf(stderr,"# of arguments in command line is not 4 or 5");
    exit(1);  }
 if((fp=fopen(*++argv, "r"))==NULL) {
    printf("can't open  %s", *argv);
    return 1;  }
 for(i=0; i<neuron_no; i++)
     for(j=0; j<neuron_no; j++) {
         fscanf(fp," %d",  &net_con[i][j]);  }
 fclose(fp);
 printf("Training ? (y/n):=");
 scanf(" %c",  &learn_flag);
 if(learn_flag == 'y') {
    printf("Input eta:=");
    scanf(" %f",  &eta);
    printf("Input gamma:=");
    scanf(" %f",  &gamma);
    printf("Input the error tolerance:=");
    scanf(" %f",  &tor_error);
    printf("Seed number(>= 65536) for PRNG:=");
    scanf(" %d",  &seed);
    printf("Display number:=");
    scanf(" %d",  &disp_no);
    if((fp=fopen(*++argv, "r"))==NULL) {
      printf("can't open  %s", *argv);
      return 1;  }
    for(i=0; i<pattern_no; i++)
       for(j=0; j<neuron_no; j++) {
          fscanf(fp," %f",  &inpattern[i][j]);  }
    fclose(fp);
    if((fp=fopen(*++argv, "r"))==NULL) {
      printf("can't open  %s", *argv);
      return 1;   }
    for(i=0; i<pattern_no; i++)
       for(j=0; j<neuron_no; j++)
          fscanf(fp," %f",  &target[i][j]);
    fclose(fp);
 }

        /*   Train the network     */

if(learn_flag=='y') {
```

```
      cycle=0;
      wgt_gen();
      while(con_flag!=1) {
        feedforward_op();
  backpropar_op();
   for(i=0; i<neuron_no; i++) {
   for(j=0; j<neuron_no; j++)
      weights[i][j]=new_weights[i][j];
   }
  cycle=cycle+1;
  if((cycle % disp_no) == 1) {
   fprintf(fp,"Iteration # %d----->",cycle);
    printf("Iteration # %d----->",cycle);
   for(i=0; i<neuron_no; i++) {
    for(j=0; j<neuron_no; j++)
       weights[i][j]=new_weights[i][j]; }

  if(cycle==300000)  con_flag = 1;
 }
printf("----- Total number of iterations ----> %d",cycle);
printf("----- Final Weights ----");
if((fp2=fopen("wgt_mom_h10_10n", "w"))==NULL) {
  printf("can't open the weight_file");
  return 1;  }
invalid_wgts=0;
for(i=0; i<neuron_no; i++) {
   for(j=0; j<neuron_no; j++) {
      if(weights[i][j]>1.0 || weights[i][j]<-1.0)
        invalid_wgts=invalid_wgts+1;
      fprintf(fp2," %.5f ",weights[i][j]);  }
 }
printf("Number of weights with invalid values =  %d",
        invalid_wgts);
fprintf(fp,"Number of weights with invalid values =  %d",
         invalid_wgts);
for(i=0;i<pattern_no;i++) {
   fprintf(fp," pattern number =  %d", i + 1);
   printf(" pattern number =  %d", i + 1);
   for(j=out_layer;j>=1; j--) {
   printf("potential[ %d][ %d]= %.3f",i,neuron_no-j-1,
           potential[i][neuron_no-j-1]);
   fprintf(fp,"potential[ %d][ %d]= %.3f",i,neuron_no-j-1,
           potential[i][neuron_no-j-1]);
      } }
  }

              /*  Test the network     */

  else {
      if((fp=fopen(*++argv, "r"))==NULL) {
        printf("can't open  %s", *argv);
        return 1;  }
      for(i=0; i<neuron_no; i++)
         for(j=0; j<neuron_no; j++)
            fscanf(fp," %f",  &weights[i][j]);
      fclose(fp);
      feedforward_op();
      printf(" ---- Results ----");
      for(i=0;i<pattern_no;i++) {
         printf(" pattern number =  %d", pattern_no);
         for(j=out_layer;j>=1; j--)
           printf("potential[ %d][ %d]= %.3f",i,neuron_no-j-1,
                   potential[i][neuron_no-j-1]);  }
  }
  fclose(fp);


          /* Feedforward operations in the DMNN   */

feedforward_op()
{
 int i,j,k,l,n1;
 float tem_inpattern[neuron_no],tem_net_ex[pattern_no][neuron_no];
 float tem_net_in[pattern_no][neuron_no];
 for(k=0; k<pattern_no ;k++) {
    for(n1=0; n1<neuron_no; n1++)
       tem_inpattern[n1]=inpattern[k][n1];
    for(i=0; i<neuron_no ; i++) {
       tem_net_ex[k][i]=1.0;
       tem_net_in[k][i]=1.0;
       for(j=0; j<neuron_no ; j++) {
          if(weights[i][j]>=0.0)
            tem_net_ex[k][i]=tem_net_ex[k][i]*(1.0-weights[i][j]
                          *tem_inpattern[j]);
```

```
        else
          tem_net_in[k][i]=tem_net_in[k][i]*(1.0+weights[i][j]
                          * tem_inpattern[j]);
       }
      tem_net_ex[k][i]=1.0-tem_net_ex[k][i];
      tem_net_in[k][i]=1.0-tem_net_in[k][i];
     tem_inpattern[i]=tem_net_ex[k][i]*(1.0-tem_net_in[k][i]);
    }
   for(n1=0; n1<neuron_no; n1++) {
     potential[k][n1]=tem_inpattern[n1];
     net_ex[k][n1]=tem_net_ex[k][n1];
     net_in[k][n1]=tem_net_in[k][n1];  }
  }
}


       /*  Back-propagation operations in the DMNN   */

backpropar_op()

{   static int iter_cnt=0;
   int i,j,k,ii,jj,kk,h;
   float delta_wgt[neuron_no][neuron_no];
   float delta_wgt2[neuron_no][neuron_no];
   float error[neuron_no],abs_error,new_sqrsum_error,sqrsum_error;
   float delta_wgt1[neuron_no][neuron_no], old_sqrsum_error=0.0;
   float old_delta_wgt[neuron_no][neuron_no], total_sse_change;
      for(i=0;i<neuron_no; i++) {
        for(j=0;j<neuron_no; j++)  {
          new_weights[i][j]=weights[i][j]; delta_wgt[i][j]=0.0;
          old_delta_wgt[i][j]=0.0; delta_wgt2[i][j]=0.0;   }
       }
      iter_cnt=iter_cnt+1;  abs_error=0.0;
      new_sqrsum_error=0.0;
      for(h=0; h<pattern_no; h++)  {
       for(i=0; i<neuron_no; i++)
          error[i]=0.0;
       ii=in_layer;
       sqrsum_error=0.0;
       for(jj=0;jj<out_layer; jj++)  {
        j=neuron_no-jj-2;
        error[j]=target[h][j]-potential[h][j];
        for(i=ii; i<neuron_no;i++) {
```

```
      delta_wgt1[j][i]=1.0;
      switch (net_con[j][i])  {
      case 1:
       if(weights[j][i] $>$= 0.0) {
         for(k=0; k<neuron_no; k++)  {
           if(weights[j][k] >= 0.0  & & k != i)
             delta_wgt1[j][i]=delta_wgt1[j][i]*(1.0-weights[j][k]
                          *potential[h][k]);
         }
       delta_wgt[j][i]=eta*error[j]*(1.0-net_in[h][j])
                    *potential[h][i]*delta_wgt1[j][i];
       delta_wgt2[j][i]=eta*error[j]*(1.0 - net_in[h][j])
                    *(1.0-net_ex[h][j]);
       }
       else  {
         for(k=0; k<neuron_no; k++)  {
          if(weights[j][k] < 0.0  & & k != i)
            delta_wgt1[j][i]=delta_wgt1[j][i]*(1.0+weights[j][k]
                          *potential[h][k]);
         }

         delta_wgt[j][i]=eta*error[j]*net_ex[h][j]*potential[h][i]
                    *delta_wgt1[j][i];
         delta_wgt2[j][i]=eta*error[j]*(1.0 - net_in[h][j])
                    *net_ex[h][j];
       } break;

       default:
         delta_wgt[j][i]=0.0;  break;
      }
      new_weights[j][i]=new_weights[j][i]+delta_wgt[j][i];
      new_weights[j][i]= new_weights[j][i] + gamma
                    *old_delta_wgt[j][i];
      old_delta_wgt[j][i] = delta_wgt[j][i];
     }
     if(error[j]<0.0) abs_error= abs_error-error[j];
     else abs_error= abs_error+error[j];
     sqrsum_error= sqrsum_error+error[j]*error[j];
    }
   sqrsum_error= sqrsum_error/2.0;
  new_sqrsum_error=new_sqrsum_error + sqrsum_error;
  kk=ii+hl_layer;
  for(j=ii; j< ii+hl_layer ; j++) {
```

```
      for(k=kk; k< kk+out_layer; k++) {
       if(potential[h][j]!=0.0)
         error[j]=error[j]+delta_wgt[k][j]*weights[k][j]
                  /potential[h][j];
       else
         error[j]=error[j]+delta_wgt2[k][j]*weights[k][j];
       }
       error[j]=error[j]/eta;
       for(i=0; i< neuron_no; i++)  {
          delta_wgt1[j][i]=1.0;
          switch (net_con[j][i])  {
          case 1:
           if(weights[j][i] $>$= 0.0) {
           for(k=0; k<neuron_no; k++)  {
           if(weights[j][k] >= 0.0  & & k != i)
            delta_wgt1[j][i]=delta_wgt1[j][i]*(1.0-weights[j][k]
                             *potential[h][k]);
            }
           delta_wgt[j][i]=eta*error[j]*(1.0-net_in[h][j])
                           *potential[h][i]*delta_wgt1[j][i];
           }
           else  {
             for(k=0; k<neuron_no; k++)  {
               if(weights[j][k] < 0.0  & & k != i)
           delta_wgt1[j][i]=delta_wgt1[j][i]*(1.0+weights[j][k]
                             *potential[h][k]);
             }
         delta_wgt[j][i]=eta*error[j]*net_ex[h][j]* potential[h][i]
                           *delta_wgt1[j][i];
           } break;
         default:
           delta_wgt[j][i]=0.0;  break;
       }
      new_weights[j][i]=new_weights[j][i]+delta_wgt[j][i];
      new_weights[j][i]=new_weights[j][i]+gamma
                        *old_delta_wgt[j][i];
      old_delta_wgt[j][i] = delta_wgt[j][i];
} } }
  if(new_sqrsum_error$<=$tor_error) con_flag=1;
  if((iter_cnt  % disp_no) == 1) {
  printf("abs_error= %f",abs_error);
          fprintf(fp,"abs_error= %f",abs_error);
          printf("  ss_error= %f",new_sqrsum_error);
          fprintf(fp,"  ss_error= %f",new_sqrsum_error);  }
}

          /*   Random generation for initial weights    */

wgt_gen()
{
  int i,j;
  static float sdnn_srand();
  for(i=0; i<neuron_no; i++) {
     if(i<in_layer || i == neuron_no - 1) {
       for (j=0; j<neuron_no; j++) {
           if(i==j)  weights[i][j]=1.0;
           else  weights[i][j]=0.0;
           printf("W[%d][%d]= %.3f",i,j,weights[i][j]);
     } }
     else {
       for(j=0; j<neuron_no; j++) {
          if(net_con[i][j]==1)  weights[i][j]=sdnn_srand();
          else  weights[i][j]=0.0;
         printf("W[%d][%d]= %.3f",i,j,weights[i][j]);
        }  }
} }

            /*   Random number generation    */

float sdnn_srand()
{   int x; float y;
     x = myrand(seed); x = x + RAND_MAX;
     y = (float) (x / (20.0 * RAND_MAX));
   return y;
}
int myrand(int_seed)
int int_seed;
{   static int ran_cnt=0,next;
   if(ran_cnt==0) next = int_seed;
   else next = next * 1103515245 + 12345;
   ran_cnt=ran_cnt + 1;
   return ((int)(next/65536)  % 32768);
}
```

APPENDIX C

VHDL Code and Corresponding Schematics

C.1 VHDL Code for DMNN Modeling

C.1.1 VHDL Package

```
package dmnn_pack is

    constant clock_period:natural:=20;
    constant N_bits: natural:=9;
    constant N_neuron: natural:=34;
    constant N_layer: natural:=2;
    constant N_pattern: natural:=35;
    constant in_unit_no: natural:=24;
    constant hidden_unit_no: natural:=4;
    constant out_unit_no: natural:=5;
    subtype neural_byte is BIT_VECTOR(N_bits downto 1);
    subtype neural_bit_vector is BIT_VECTOR(N_neuron down 1);
    type neural_bit_matrix is array(natural range 1 to N_neuron)
          of neural_bit_vector;
    type neural_array is array(natural range 1 to N_neuron) of
          neural_byte;
    type out_state_array is array(natural range 1 to out_unit_no)
          of neural_byte;
    type neural_matrix is array(natural range 1 to N_neuron) of
          neural_array;
    type pattern_matrix is array(natural range 1 to N_pattern) of
          neural_array;
    function LFSR_fun(rng_value, rng_type: neural_byte) return bit;
    function Wired_Or(input:neural_bit_matrix) return
                neural_bit_vector;
    function byte_val(input:neural_byte) return natural;
    function int_to_byte(int_value: natural) return neural_byte;
    function byte_to_real(in_byte:neural_byte) return real;
    function real_to_byte(in_real:real) return neural_byte;

end dmnn_pack;

package body dmnn_pack is

function LFSR_fun(rng_value, rng_type: neural_byte) return bit is
      variable feedback_bit: bit:='0';
begin
    for i in (N_bits - 1) downto 1 loop
      feedback_bit:=feedback_bit xor (rng_value(i) and rng_type(i));
    end loop;
    return feedback_bit;
end;

function Wired_Or(input:neural_bit_matrix) return neural_bit_vec-
          tor is
            variable ored_out: neural_bit_vector;
begin
    loop_out:
    for i in 1 to N_neuron loop
      loop_in:
      for j in 1 to N_neuron loop
        if(input(i)(j)='1') then ored_out(i):='1';
        end if;
      end loop loop_in;
    end loop loop_out;
    return ored_out;
end;

function byte_val(input:neural_byte) return natural is
      variable int_val: natural:=0;
begin
    for i in (N_bits - 1) downto 1 loop
      if(input(i)='1') then int_val:=int_val+2**(i-1);
      end if;
    end loop;
    return int_val;
end;

function int_to_byte(int_value: natural) return neural_byte is
      variable int_val: natural;
      variable byte_val: neural_byte;
begin
   int_val:=int_value;
```

```
        for i in (N_bits - 1) downto 1 loop
          if(int_val/(2**(i-1)) >= 1) then
            byte_val(i):='1';
            int_val:=int_val - (2**(i-1));
          else byte_val(i):= 0;
        end if;
        end loop;
        return byte_val;
end;

function byte_to_real(in_byte:neural_byte) return real is
          variable real_val: real:=0.0;
          variable cnt_1: natural:=0;
begin
         for i in (N_bits - 1) downto 1 loop
           if(in_byte(i)='1') then
             real_val:=real_val+1.0/(2.0**(N_bits - i));
             cnt_1:=cnt_1+1;
           end if;
         end loop;
         if(cnt_1=N_bits-1) then real_val:=1.0;
         end if;
         if(in_byte(N_bits)='1') then real_val:=real_val*(-1.0);
         end if;
         return real_val;
end;
function real_to_byte(in_real:real) return neural_byte is
          variable real_val: real:=0.0;
          variable byte_val: neural_byte;
begin
        real_val:=in_real;
        if(real_val<0.0) then
          byte_val(N_bits):='1';
          real_val:=real_val*(-1.0);
        end if;
        for i in (N_bits - 1) downto 1 loop
          if(real_val/(1.0/2.0**(N_bits-i)) >= 1.0) then
            byte_val(i):='1';
            real_val:=real_val - (1.0/2.0**(N_bits-i));
          else  byte_val(i):= 0;
          end if;
        end loop;
        if(real_val/(1.0/2.0**N_bits) >= 1.0) then byte_val(1):='1';
        end if;
        return byte_val;
    end;
end dmnn_pack;
```

C.1.2 DMNN Coprocessor

```
use work.dnn_pack.all;
entity dmnn_coprocessor is
end dmnn_coprocessor;

architecture testbench of dmnn_coprocessor is

    component clock
          port(start: in bit;
               clk_out: out bit);
    end component;
    component step_cnt
          port(step_clk: in bit;
               new_step: out bit);
    end component;
    component dnn_net
          port(net_load,net_clk,net_enable,next_period: in bit;
               new_pattern: in neural_array;
               weight_in: in neural_matrix;
               net_state_out: out out_state_array);
    end component;
    component control_unit
          port(ctl_enable,ctl_prd: in bit;
               net_load,ram_rw,clk_on,net_on,ram_on: out bit);
    end component;
```

```
component RAM
        port(ram_enable,r_w: in bit;
            state_in: in out_state_array;
            new_pattern: out neural_array;
            weight_out: out neural_matrix);
end component;
type out_bits is array(natural range 1 to out_unit_no) of bit;
signal out_light: out_bits;
signal run: bit:='0';
signal clk_sig: bit:='0';
signal net_load_sig: bit:='0';
signal net_ena_sig: bit:='0';
signal clk_ena_sig: bit:='0';
signal new_prd_sig: bit:='0';
signal new_pattern_sig: neural_array;
signal net_weight_in: neural_matrix;
signal net_state_sig: out_state_array;
signal ram_rw_sig: bit;
signal ram_ena_sig: bit;
signal ram_net_state: pattern_matrix;

for all: clock use entity work.clock(clk_behavior);
for all: step_cnt use entity work.step_cnt(step_cnt_dnn);
for all: control_unit use entity work.control_unit(behavior);
for all: dnn_net use entity work.dnn_net(behavior);
for all: RAM use entity work.RAM(behavior);

begin
   out_light(1)<=net_state_sig(1)(N_bits-1);
   out_light(2)<=net_state_sig(2)(N_bits-1);
   out_light(3)<=net_state_sig(3)(N_bits-1);
   out_light(4)<=net_state_sig(4)(N_bits-1);
   out_light(5)<=net_state_sig(5)(N_bits-1);

   run <= '1' after 5 ns;
   Clk_Blk: clock port map (clk_ena_sig, clk_sig);
   Step_Blk: step_cnt port map (clk_sig, new_prd_sig);
   Control_Blk: control_unit port map (run,new_prd_sig,
                    net_load_sig, ram_rw_sig,clk_ena_sig,
                        net_ena_sig,ram_ena_sig);
      Dnn_net_BLK: dnn_net port map (net_load_sig, clk_sig,
                        net_ena_sig,  new_prd_sig,new_pattern_sig,
                        net_weight_in,net_state_sig);
      RAM_BLK: RAM port map (ram_ena_sig,ram_rw_sig,
                    net_state_sig, new_pattern_sig,net_weight_in);
end testbench;
```

C.1.3 DMNN: Network

```
use work.dmnn_pack.all;

entity dmnn_net is
        port(
            net_load, net_clk, net_enable, next_period: in bit;
            new_pattern: in neural_array;
            weight_in: in neural_matrix;
            net_state_out: out out_state_array);
end dmnn_net;

use std.textio;
use std.simulator_standard.terminate;

architecture behavior of dmnn_net is

    component synapse
        port(syn_load, syn_clk: in bit;
            syn_weight, syn_ran_in: in neural_byte;
            action_stream: in bit;
            ex_net_out, in_net_out: out bit;
    end component;
    component neuron_body
        port(ex_net_input, in_net_input, body_load, body_clk,
            step_cnt: in bit;
            init_state: in neural_byte;
            init_random: in neural_byte;
            output_stream: out bit;
            neuron_output: out neural_byte);
    end component;
    signal clk: bit;
    signal syn_ran_in: neural_matrix;
    signal body_ran_in, net_output: neural_array;
    signal out_stream, in_wired_or, ex_wired_or:
            neural_bit_vector;
```

```
    signal ex_or, in_or: neural_bit_matrix;
    for all: synapse use entity work.synapse(synap_dmnn);
    for all: neuron_body use entity work.neuron_body
          (nbody_dmnn);
begin

    Main_blk: block(net_enable='1')
      begin
        clk<= guarded net_enable and net_clk;
        ex_wired_or<=guarded Wired_Or(ex_or);
        in_wired_or<=guarded Wired_Or(in_or);
    end block

  Init_RNG: process

  type int_array is array(natural range 1 to N_neuron) of integer;
  type int_matrix is array(natural range 1 to N_neuron) of
       int_array;
  file F_sran_in: textio.TEXT is in "syn_ran_input";
  file F_bran_in: textio.TEXT is in "body_ran_input";
  variable L3, L4: textio.LINE; variable in_vec: neural_array;
  variable in_weight: neural_matrix; variable in_bran: int_array;
  variable in_sran: int_matrix; variable new_iter: natural:=0;
      begin
          L3 := new STRING' (""); L4 := new STRING' ("");
          for i in 1 to N_neuron loop
              textio.readline(F_bran_in, L4);
                  textio.read(L4, in_bran(i));
              body_ran_in(i)<=int_to_byte(in_bran(i));
          end loop;
          for i in 1 to in_unit_no loop
              for j in 1 to N_neuron loop
                 if(i=j)then
                   textio.readline(F_sran_in, L3);
                       textio.read(L3, in_sran(i)(j));
                 else
                   in_sran(i)(j):=0;
                 end if;
                 syn_ran_in(i)(j)<=int_to_byte(in_sran(i)(j));
              end loop;
          end loop;
          for i in (in_unit_no+1) to N_neuron loop
              for j in 1 to N_neuron loop
                   textio.readline(F_sran_in, L3);
                      textio.read(L3, in_sran(i)(j));
                    syn_ran_in(i)(j)<=int_to_byte(in_sran(i)(j));
                  end loop;
              end loop;
              wait for 100 ms;
              terminate;
      end process;

  Input_Layer: for i in 1 to in_unit_no generate
   Input_Synap_element:
      synapse port map(net_load,clk,weight_in(i)(i),
       syn_ran_in(i)(i), out_stream(i),ex_or(i)(i),in_or(i)(i));

  Input_Body_element:
          neuron_body port map(ex_wired_or(i),in_wired_or(i),
                  net_load,clk,next_period,new_pattern(i),
                 body_ran_in (i),out_stream(i),net_output(i));

  end generate Input_Layer;
  Hidden_Layer: for j in (in_unit_no+1) to (in_unit_no+
                                      hidden_unit_no)  generate
       H_SYNAPSE_ARRAY: for i in 1 to in_unit_no generate
         Hidden_Synap_element:
                synapse port map (net_load,clk,weight_in(j)(i),
                                 syn_ran_in(j)(i),out_stream(i),
                                  ex_or(j)(i),in_or(j)(i));

      end generate H_SYNAPSE_ARRAY;
      H_Threshold_synapse:
            synapse port map(net_load,clk,weight_in(j)(N_neuron),
                  syn_ ran_in(j)(N_neuron),out_stream(N_neuron),
                              ex_or(j)(N_neuron),in_or(j)(N_neuron));
      Hidden_Body_element:
              neuron_body port map(ex_wired_or(j),in_wired_or(j),
                          net_load,clk,next_period,new_pattern(j),
                           body_ran_in(j),out_stream(j),net_output(j));
  end generate Hidden_Layer;

  Output_Layer: for k in (in_unit_no+hidden_unit_no+1) to
          (in_unit_no+hidden_unit_no+out_unit_no) generate
     O_SYNAPSE_ARRAY:
        for j in (in_unit_no+1) to (in_unit_no+hidden_unit_no)
             generate  Output_Synap_element:
```

```
                    synapse port map(net_load,clk,weight_in(k)(j),
            syn_ran_in(k)(j),out_stream(j),ex_or(k)(j),in_or(k)(j));
        end generate O_SYNAPSE_ARRAY;
    O_Threshold_synapse: synapse port map (net_load,clk,
          weight_in(k)(N_neuron),syn_ran_in(k)(N_neuron),
                out_stream(N_neuron), ex_or(k)(N_neuron),
                in_or(k)(N_neuron));
    Output_Body_element:
              neuron_body port map(ex_wired_or(k),
                  in_wired_or(k),net_load,clk,next_period,
                    new_pattern(k), body_ran_in(k),out_stream(k),
                     net_state_out(k-in_unit_no-hidden_unit_no));
    end generate Output_Layer;
  Threshold_synapse_element: synapse
            port map (net_load,clk,weight_in(N_neuron)
                (N_neuron),syn_ran_in(N_neuron)(N_neuron),
          out_stream(N_neuron),ex_or(N_neuron)(N_neuron),
                in_or(N_neuron)(N_neuron));
  Threshold_Body_element: neuron_body
         port map (ex_wired_or(N_neuron),
             in_wired_or(N_neuron),net_load,clk,next_period,
             new_pattern(N_neuron), body_ran_in(N_neuron),
             out_stream(N_neuron), net_output(N_neuron));
end behavior;
```

C.1.4 Control Unit

```
use work.dmnn_pack.all;
entity control_unit is
      port(
          ctl_enable, ctl_prd: in bit;
          net_load, ram_rw, clk_on, net_on, ram_on: out bit;
end control_unit;

use std.simulator_standard.terminate;
architecture behavior of control_unit is

     signal net_switch, ram_switch, new_pattern, new_iter,
            clk_switch: bit:='0';

     signal con_in, stop_sig: bit:='0';

begin
          stop_sig <= con_in after 50 ns;

          IO_control: block(ctl_enable='1')
              begin
                   ram_on<=guarded ram_switch;
                   net_on<=guarded net_switch;
                   clk_on<=guarded clk_switch;
          end block;
          Termination_control: process(con_in)
          begin
                   if(stop_sig='1' and not stop_sig'stable) then
                     terminate;
                   end if;
          end process;
main_control: process
begin
        wait for 5 ns;
        net_switch<='1'; ram_rw<='1'; ram_switch<='1';
        wait for 5 ns;
        clk_switch<='1'; net_load<='1', '0' after 20 ns;
        ram_switch<='0';
          wait until new_pattern='1' and not new_pattern'stable;
        wait for 25 ns;
        clk_switch<='0'; ram_rw<='0'; ram_switch<='1';
        wait for 10 ns;
        net_switch<='0'; clk_switch<='0'; wait for 40 ns;
        ram_switch<='0';
  end process;
  Pattern_control: process
            variable prd_cnt: natural:=0;
  begin
       wait until ctl_prd='1' and not ctl_prd'stable;
       prd_cnt:=prd_cnt + 1;
       if(prd_cnt=N_layer) then
         new_pattern<='1','0' after 20 ns;
         prd_cnt:=0;
       end if;
  end process;

  DNS_control: process
```

```
            variable ptn_cnt: natural:=0;
    begin
        wait until new_pattern='1' and not new_pattern'stable;
          ptn_cnt:=ptn_cnt + 1;
          if(ptn_cnt=N_pattern) then
           con_in<='1', '0' after 50 ns;
           ptn_cnt:=0;
          end if;
    end process;

end behavior;
```

C.1.5 Memory

```
use work.dmnn_pack.all;
entity RAM is
    port(ram_enable, r_w: in bit;
         state_in: in out_state_array;
         new_pattern: out neural_array;
         weight_out: out neural_matrix);
end RAM;

use std.textio;
use std.simulator_standard.terminate;
architecture behavior of RAM is

   type real_array is array(natural range 1 to N_neuron) of real;
   type real_matrix is array(natural range 1 to N_pattern) of
         real_array;
   type real_neural_matrix is array(natural range 1 to N_neuron)
         of real_array;
   type real_out_array is array(natural range 1 to out_unit_no)
         of real;
   type real_out_matrix is array(natural range 1 to N_pattern)
         of real_array;
   signal pattern_mem, state_mem: pattern_matrix;
   signal weight_mem: neural_matrix;
 begin
   Initialization: process
      file F_in1: textio.TEXT is in "5_digit_in_pattern";
      file F_in2: textio.TEXT is in "5_digit_10p_weight8";
      variable L1, L2: textio.LINE;
       variable in_pattern: real_matrix;
       variable in_weight: real_neural_matrix;
 begin
      L1 := new STRING' (""); L2 := new STRING' ("");
      for i in 1 to N_pattern loop
          for j in 1 to N_neuron loop
             textio.readline(F_in1, L1);
                textio.read(L1, in_pattern(i)(j));
             pattern_mem(i)(j)<=real_to_byte(in_pattern(i)(j));
          end loop;
      end loop;
      for i in 1 to in_unit_no loop
          for j in 1 to N_neuron loop
             if(i=j)then
                   in_weight(i)(j):=1.0;
             else in_weight(i)(j):= 0.0;
             end if;
             weight_mem(i)(j)<=real_to_byte(in_weight(i)(j));
          end loop;
      end loop;
      for i in (in_unit_no+1) to N_neuron loop
          for j in 1 to N_neuron loop
             textio.readline(F_in2, L2);
                textio.read(L2, in_weight(i)(j));
             weight_mem(i)(j)<=real_to_byte(in_weight(i)(j));
          end loop;
      end loop;
      wait for 100 ms;
      terminate;
end process;

Memory_rw: process
       variable pattern_cnt1, pattern_cnt2: natural:=0;
       file F_out: textio.TEXT is out "5_digit_out_state8";
       variable L3: textio.LINE;
       variable state_out: real_out_matrix;
 begin
      L3 := new STRING' ("");
      wait until ram_enable='1' and not ram_enable'stable;
      if(r_w='0') then
       pattern_cnt1:=pattern_cnt1+1;
```

```
            for i in 1 to out_unit_no loop
                state_mem(pattern_cnt1)(i)<=state_in(i);
            end loop;
           wait for 5 ns;
           if(pattern_cnt1=N_pattern) then
              for i in 1 to N_pattern loop
                 for j in 1 to out_unit_no loop
                      state_out(i)(j):=byte_to_real(state_mem(i)(j));
                      textio.write(L3, state_out(i)(j));
                      textio.writeline(F_out, L3);
                 end loop;
              end loop;
            pattern_cnt1:=0;
           end if;
           end if;
           if(r_w='1') then
            pattern_cnt2:=pattern_cnt2+1;
            for i in 1 to N_neuron loop
             new_pattern(i)<=pattern_mem(pattern_cnt2)(i);
             for j in 1 to N_neuron loop
               weight_out(i)(j)<=weight_mem(i)(j);
             end loop;
            end loop;
          if(pattern_cnt2=N_pattern) then
           pattern_cnt2:=0;
          end if; end if;
      end process;
 end behavior;
```

C.1.6 Synaptic Element

```
use work.dmnn_pack.all;
entity synapse is port(syn_loadi, syn_clk: in bit;
                       syn_weight, syn_ran_in: in neural_byte;
                       action_stream: in bit;
                       ex_net_out, in_net_out: out bi)t;
end synapse;
architecture synap_dmnn of synapse is
        component RGT
            port(rgt_load, rgt_clk: in bit;
                 rgt_in, rgt_out: in neural_byte;
        end component;
        component RNG
            port(rng_in: in neural_byte;
                 rng_load, rgn_clk: in bit;
                 rng_out: out neural_byte);
        end component;
        component CMP
            port(cmp_in1, cmp_in2: in neural_byte;
                 cmp_out: out bit);
        end component;
        signal ran_num, weight_out: neural_byte;
        signal com_out: bit;
        for all: RGT use entity work.RGT(RGT_dmnn);
        for all: RNG use entity work.RNG(RNG8_t1);
        for all: CMP use entity work.CMP(CMP_dmnn);

 begin
        RNG_1: RNG port map (syn_ran_in,syn_load,syn_clk,
                             ran_num);
        CMP_1: CMP port map (weight_out, ran_num, com_out);
        WT_REG: RGT port map (syn_load, syn_clk, syn_weight,
        Synapse_operation:          weight_out);
            process(weight_out(N_bits),com_out, action_stream)
                 variable ex_andout, in_andout: bit;
             begin
                 ex_andout:= com_out and action_stream
                             and not weight_out(N_bits);
                 in_andout:= com_out and action_stream
                             and  weight_out(N_bits);
                 ex_net_out<= ex_andout after 4 ns;
                 in_net_out<= in_andout after 4 ns;

        end process;
 end synap_dmnn;
```

C.1.7 Neuron Body Element

```
 use work.dmnn_pack.all;
```

```
entity neuron_body is
    port(ex_net_input, in_net_input, body_clk, step_cnt, : in bit;
        init_state, init_random: in neural_byte;
        output_stream: out bit;
        neuron_output: out neural_byte);
end neuron_body;

architecture Nbody_dmnn of neuron_body is
    component MUX
            port(mux_in1, mux_in2: in neural_byte;
                mux_sel: in bit;
                mux_out: out neural_byte);
    end component;
    component RGT
            port(rgt_load, rgt_clk: in bit;
                rgt_in: in neural_byte;
                rgt_out: out neural_byte);
    end component;
    component CNT
            port(cnt_in, cnt_clk, reset: in bit;
                cnt_out: out neural_byte);
    end component;
    component RNG
            port(rng_in: in neural_byte;
                rng_load, rng_clk: in bit;
                rng_out: out neural_byte);
    end component;
    component CMP
            port(cmp_in1, cmp_in2: in neural_byte;
                cmp_out: out bit);
    end component;
    signal ran_num, counter_out, mux_out_sig: neural_byte;
    signal current_state: neural_byte;
    signal buffer_load, up_sig: bit;
    for all: MUX use entity work.MUX(mux_dmnn);
    for all: RGT use entity work.RGT(RGT_dmnn);
    for all: CNT use entity work.CNT(CNT_dmnn);
    for all: RNG use entity work.RNG(RNG8_t2);
    for all: CMP use entity work.CMP(CMP_dmnn);
begin
    net_input: process(ex_net_input, in_net_input)
    begin
        p_sig <= ex_net_input and not in_net_input after 4 ns;
    end process;
    next_step: process(body_load, step_cnt)
        begin
            buffer_load <= body_load or step_cnt after 4 ns;
    end process;
    MUX_1: MUX port map (init_state, counter_out,body_load,
                                    mux_out_sig);
    BUFFER1: RGT port map (buffer_load, body_clk,
                                    mux_out_sig, current_state);
    COUNTER: CNT port map (up_sig, body_clk, step_cnt,
                                    counter_out);
    RNG_2: RNG port map (init_random,body_load,body_clk,
                                    ran_num);
    CMP_1: CMP port map (current_state,ran_num,
                                    output_stream);
    neuron_output <= current_state;

end Nbody_dmnn;
```

C.1.8 Register

```
use work.dmnn_pack.all;
entity RGT is
    port(rgt_load: in bit;
        rgt_clk: in bit;
        rgt_in: in neural_byte;
        rgt_out: out neural_byte);

end RGT;

architecture RGT_dmnn of RGT is
  begin
    process(rgt_clk)
    begin
        if(rgt_clk = '1' and rgt_load = '1' and not rgt_clk'stable )
            then rgt_out <= rgt_in after 10 ns;
        end if;
    end process;
end RGT_dmnn;
```

C.1.9 Random Number Generator

```
use work.dmnn_pack.all;
entity RNG is
                port( rng_in: in neural_byte;
                    rng_load, rng_clk: in bit;
                    rng_out: out neural_byte);
          end RNG;

------- RNG8_t1 is a type #1 RNG whose configuration is -------
-------- 435 : 100011101=> "001110001"              ------------

 architecture RNG8_t1 of RNG is

    begin
        process(rng_clk)
          variable tmp_buffer: neural_byte;
          variable feedback_sig: bit;
          constant rng_type: neural_byte:= "001110001";
        begin
          i f(rng_clk = '1' and not rng_clk'stable) then
               if rng_load = '1' then  tmp_buffer:=rng_in;
             else feedback_sig:= LFSR_fun(tmp_buffer,rng_type);
                   loop_2: for i in 1 to (N_bits - 2) loop
                         tmp_buffer(i):=tmp_buffer(i+1);
                   end loop loop_2;
                    tmp_buffer(N_bits - 1):=feedback_sig;
          end if; end if;
          rng_out <= tmp_buffer after 10 ns;
       end process;
 end RNG8_t1;

-------- RNG8_t2 is a type 2 RNG whose configuration is -------
---------   765: 111110101=> "001011111"              ------------

architecture RNG8_t2 of RNG is
   begin
       process(rng_clk)
         variable tmp_buffer: neural_byte;
         variable feedback_sig: bit;
          constant rng_type: neural_byte:= "001011111";
       begin
          if(rng_clk = '1' and not rng_clk'stable) then
            if rng_load = '1' then  tmp_buffer:=rng_in;
            else   feedback_sig:= LFSR_fun(tmp_buffer,rng_type);
                   loop_2: for i in 1 to (N_bits - 2) loop
                         tmp_buffer(i):=tmp_buffer(i+1);
                   end loop loop_2;
                    tmp_buffer(N_bits - 1):=feedback_sig;

          end if; end if;
          rng_out <= tmp_buffer after 10 ns;
       end process;
end RNG8_t2;
```

C.1.10 Counter

```
use work.dmnn_pack.all;
entity CNT is
     port(cnt_in, cnt_clk, reset: in bit;
          cnt_out: out neural_byte);
end CNT;

architecture CNT_dmnn of CNT is
     begin
       cnt_up_process:  process(cnt_clk)
                variable up_counter: natural:=0;
             begin
               if (cnt_clk='1' and not cnt_clk'stable and reset='1')
                then  up_counter:=0;
                      cnt_out <= int_to_byte(up_counter) after 10 ns;
              elsif (cnt_clk='1' and not cnt_clk'stable and cnt_in='1')
                then   up_counter:=up_counter+1;
                       cnt_out <= int_to_byte(up_counter) after 10 ns;
                end if;
            end process;
    end CNT_dmnn;
```

C.1.11 Digital Comparator

```
use work.dmnn_pack.all;
entity CMP is
      port(cmp_in1, cmp_in2: in neural_byte;
          cmp_out: out bit);
end CMP;
architecture CMP_dmnn of CMP is
    begin
      compare: process(cmp_in1, cmp_in2)
                    variable in1, in2:natural;
          begin
            in1 := byte_val(cmp_in1);  in2 := byte_val(cmp_in2);
            if(in1 < in2 or in2 = 0) then
                  cmp_out <= '0' after 10 ns;
            else  cmp_out <= '1' after 10 ns;
            end if;
     end process;
end CMP_dmnn;
```

C.1.12 Mulplexer

```
use work.dnn_pack.all;
entity MUX is
            port(mux_in1,mux_in2: in neural_byte;
                mux_sel: in bit;
                mux_out: out neural_byte);
end MUX;
architecture mux_DNN of MUX is
   begin
      mux_process: process(mux_in1,mux_in2,mux_sel)
          begin
                if mux_sel = '1' then mux_out <= mux_in1 after 4 ns;
                else mux_out <= mux_in2 after 4 ns;
                end if;
         end process;
end mux_DNN;
```

C.1.13 Iteration Counter

```
use work.dmnn_pack.all;
entity step_cnt is
     port(step_clk: in bit;
          new_step: out bit);
end step_cnt;
architecture step_cnt_dmnn of step_cnt is
    begin
        process
          variable cnt: natural:=0;
        begin
          wait until step_clk='1' and not step_clk'stable;
          cnt:=cnt+1;
          if(cnt=(2**(N_bits-1)+1)) then
               new_step <= '1' after 10 ns, '0' after 30 ns;
          end if;
          if(cnt=(2**(N_bits-1)+2)) then cnt:=0;
           end if;
        end process;
 end step_cnt_dmnn;
```

C.1.14 Clock Generator

```
use work.dmnn_pack.all;
entity clock is
     port( start: in bit;
          clk_out: out bit);
end clock;
architecture clk_behavior of clock is
  begin
      process
        begin
         if(start='1') then clk_out<='1';
            wait for clock_period/2 *1 ns; clk_out<='0';
            wait for clock_period/2 *1 ns;
          else wait for clock_period*1 ns;
         end if;
      end process;
end clk_behavior;
```

C.2 Figures

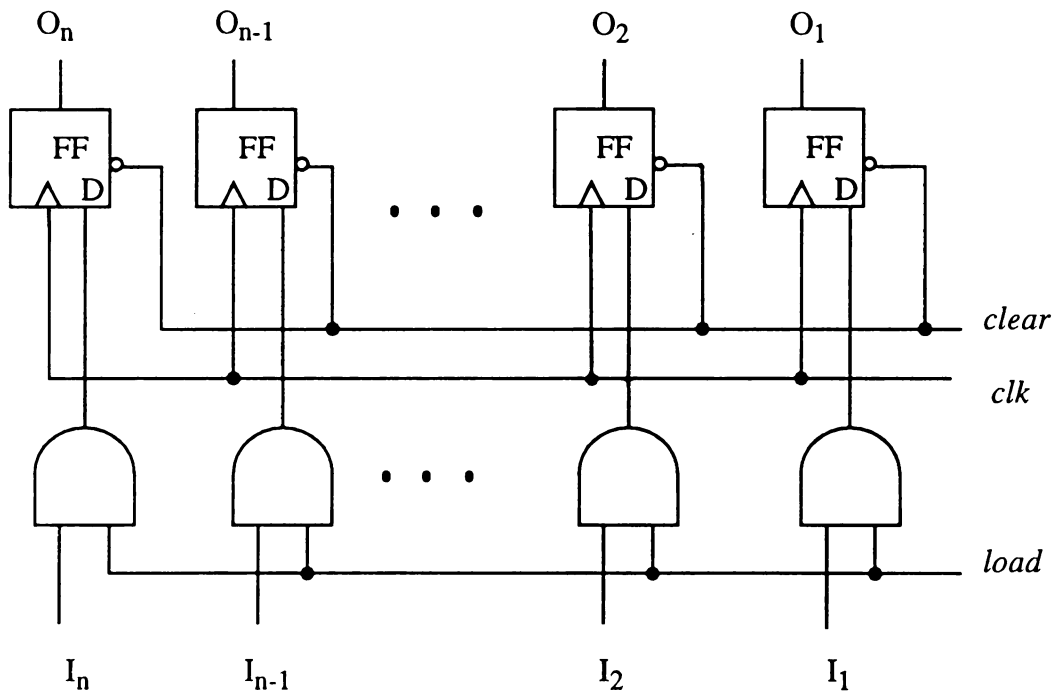


Figure C.1. An *n*-bit register with parallel load.

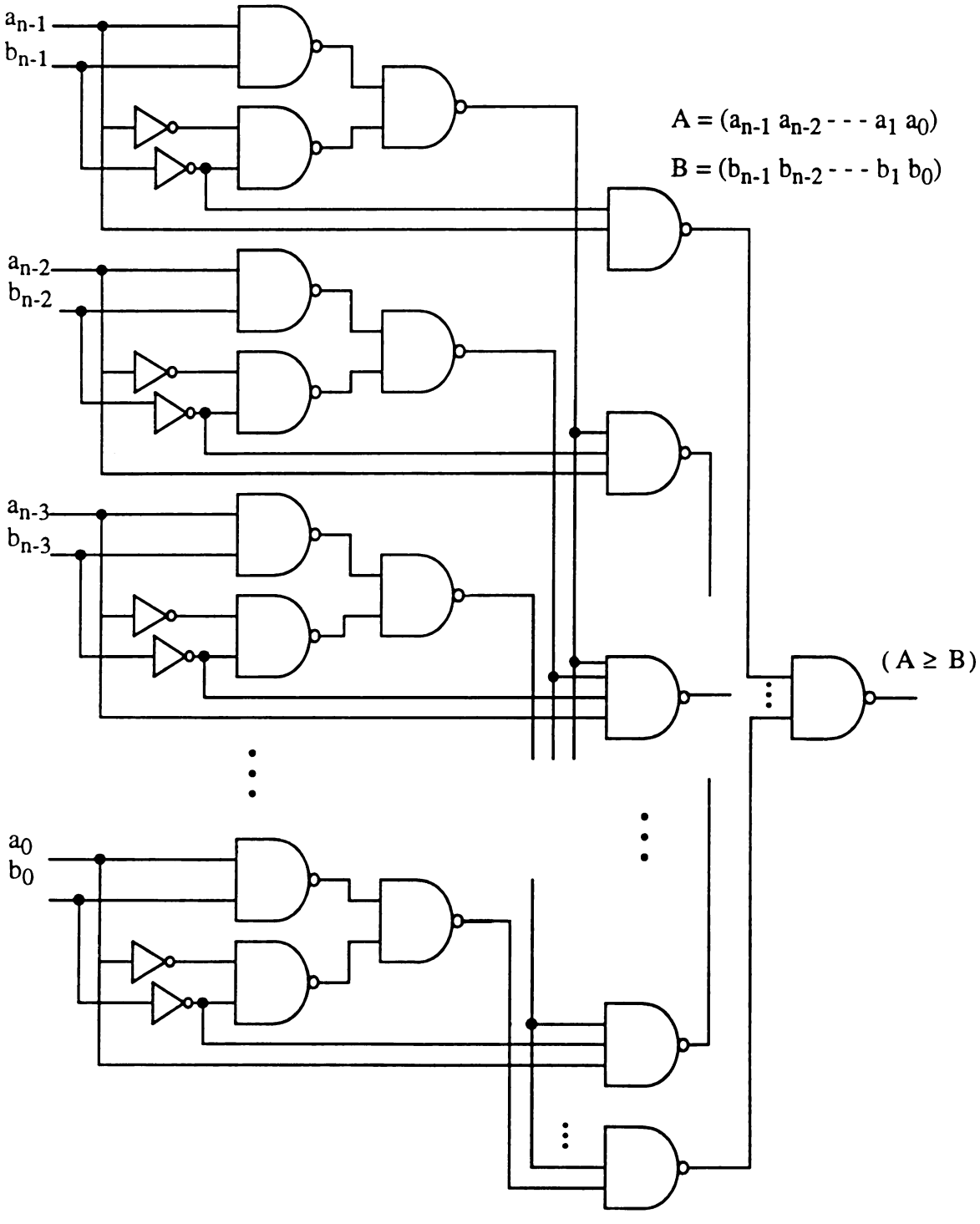


Figure C.2. An *n*-bit magnitude comparator.

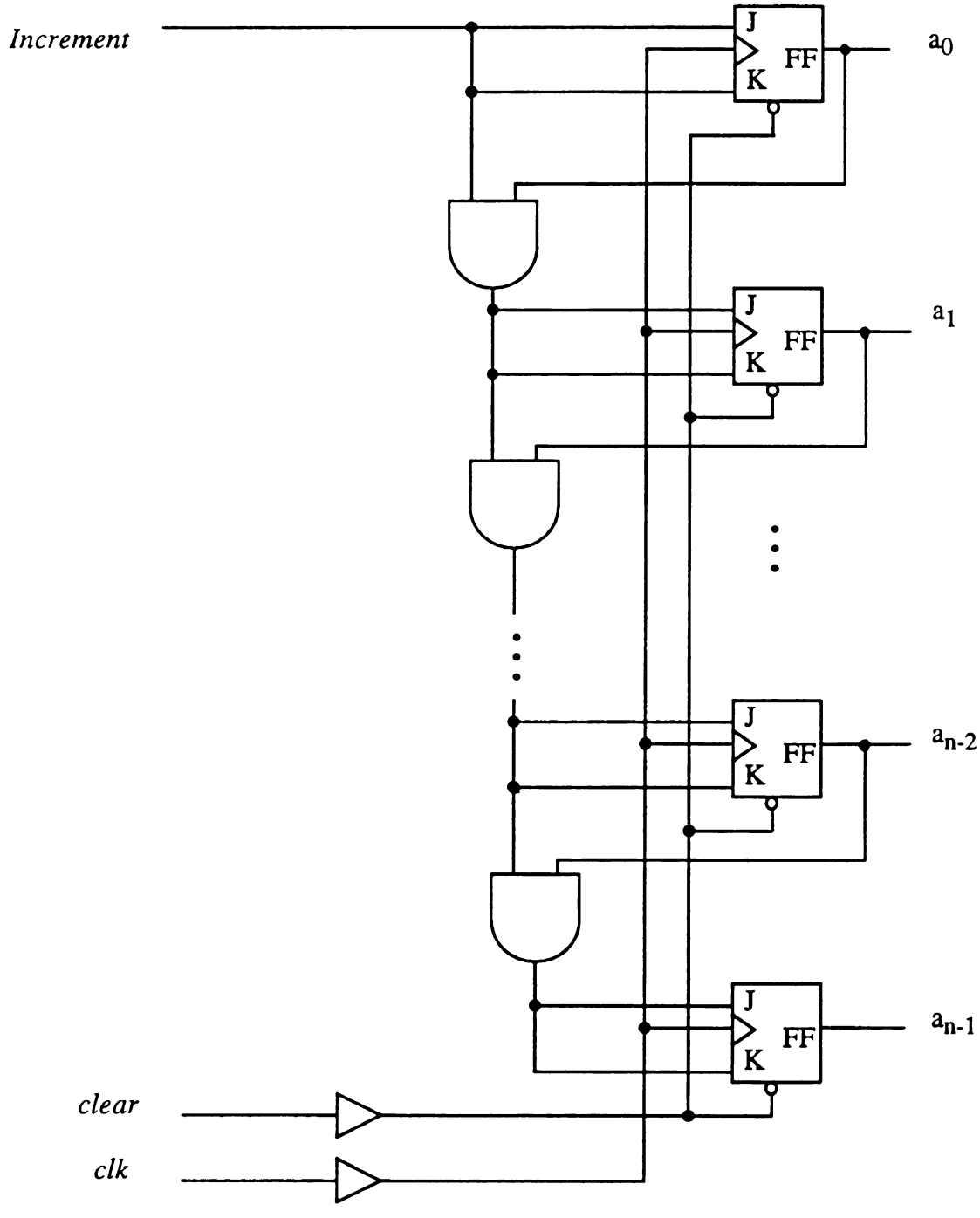


Figure C.3. An *n*-bit up-counter.

APPENDIX D

Input Data for DMNN Binary Classifiers

Typical exmaples of network configuration, test patterns, and synaptic weights used for the XOR, 8-to-3 encoding, 5-digit classification, and 10-digit classification problems are listed. For network configuration and synaptic weights, the value in row i and column j represents the connectivity and synaptic weight between neurons i and j, respectively. The *ith* element of a test pattern vector is the value applied to the *ith* neuron in the input layer.

D.1 XOR Problem

- Network configuration:
```
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
```

- Test patterns:
```
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
```

- Synaptic weights:
```
1.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0
0.91524 0.92400 0.0 0.0 0.0 0.02375
0.34412 0.34811 0.0 0.0 0.0 -0.00001
0.0 0.0 0.53978 -0.68425 0.0 0.44300
0.0 0.0 0.0 0.0 0.0 1.0
```

D.2 8-to-3 Encoding Problem

- Network configuration:
```
1 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 1 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 1 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 1 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
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 0 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 0 1
```

- Test pattern:
```
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 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 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 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 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 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 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 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
```

- Synaptic weight:
```
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 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 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 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 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 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 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 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
-0.00260 -0.00178 0.32771 0.32397 -0.00057 -0.00112 0.28613
0.28223 0.0 0.0 0.0 0.0 0.0 0.0 0.00012
-0.00250 0.31066 -0.00164 0.31067 -0.00029 0.30321 -0.00072
0.30226 0.0 0.0 0.0 0.0 0.0 0.0 0.00007
-0.04424 -0.02010 0.01493 -0.00085 0.29708 0.28673 0.32232
0.31286 0.0 0.0 0.0 0.0 0.0 0.0 0.05632
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -0.04127 0.02067 0.61525 0.0 0.0 0.0
0.43104
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.58922 0.01009 0.08112 0.0 0.0 0.0
0.44059
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.00055 0.59283 0.01644 0.0 0.0 0.0
0.44809
0.0 0.0 0.0 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
```

D.3 5-digit Classification Problem

For 5-digit and 10-digit classification problems, the test patterns consist of ideal digits, 10 % noisy patterns, and 20 % noisy patterns. For each digit in test patterns, one ideal digit, three 10% noisy patterns, and three 20 % noisy patterns are listed in a row.

- Network configuration:

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

- Test patterns:

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

- Synaptic weights when the DMNN is trained with ideal digits:

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 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0
0.0 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 0.0 0.0 0.0 0.0
0.0 0.0 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 0.0 0.0 0.0
0.0 0.0 0.0 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 0.0 0.0
0.0 0.0 0.0 0.0 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 0.0
0.0 0.0 0.0 0.0 0.0 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
0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 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 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 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00328 0.13224 0.03847 0.02044 -0.00548 0.07845 0.02449
0.02303 0.00422 -0.03103 0.12277 -0.01014 0.12590 0.10676 -
0.01083 0.20680 0.05309 -0.06927 0.04795 0.18354 0.11931
0.16356 0.04998 0.00308 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00136

-0.05893 0.03705 0.00838 0.07205 0.16624 0.03670 0.15082 -
0.01185 0.20633 -0.06129 0.06269 -0.02770 0.16211 -0.01037 -
0.00857 0.02403 0.22697 -0.03821 0.12019 0.02411 0.00392
0.03719 0.04337 0.05853 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.02028
-0.01628 0.06386 0.05834 0.06659 0.13125 0.05654 0.01117
0.13257 0.02745 0.00933 -0.10847 0.12733 0.04385 -0.06614
0.06932 -0.03110 0.00816 0.26632 -0.05490 -0.12803 0.07732
0.02874 0.05797 0.18871 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.08037
0.01503 0.04847 0.04788 0.01860 0.00067 0.05150 0.19844 -
0.03957 -0.03392 0.05018 0.16888 -0.10619 0.03350 0.02692
0.10805 0.02773 0.03182 -0.04625 0.11554 -0.02728 0.15753
0.04848 0.05429 0.14014 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.05058
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.22753 0.34160 0.05412 -0.25289 0.0 0.0
0.0 0.0 0.0 0.36682
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.19970 0.24424 0.17877 0.44400 0.0 0.0
0.0 0.0 0.0 0.09193
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 -0.07011 -0.13617 0.44648 -0.03321 0.0 0.0
0.0 0.0 0.0 0.39364
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.29237 -0.33505 -0.21928 -0.00165 0.0 0.0
0.0 0.0 0.0 0.54798
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 -0.39267 0.01086 -0.30371 -0.10424 0.0 0.0
0.0 0.0 0.0 0.71870
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 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

- Synaptic weights when the DMNN is trained with ideal and 10 % noisy patterns:

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 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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
0.07840 0.07874 0.10147 0.01799 0.18720 0.17039 -0.07353

0.26599 -0.02935 -0.00001 -0.00868 0.13180 -0.20208 0.27351 -
0.05087 -0.01591 0.14653 0.20231 0.00004 0.09641 -0.04758
0.18747 0.00001 0.26337 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -
0.00001
-0.07860 0.16221 0.01949 0.09311 -0.03029 -0.00422 0.13984 -
0.06780 -0.01893 -0.10872 0.19052 -0.18966 -0.16158 -0.14315
0.00221 -0.04125 -0.12754 -0.00001 0.18903 0.00661 0.14060 -
0.00758 -0.00002 0.18032 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -
0.00002
0.06153 0.31482 0.06430 0.02372 0.04495 0.05690 -0.02502
0.08432 -0.03299 -0.10703 0.15937 -0.02753 0.42001 0.03077 -
0.02721 -0.02125 -0.01576 0.07280 0.29793 0.09075 0.00194
0.05473 -0.00001 -0.26931 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.01547 -0.00002 0.00535 0.03131 -0.00003 -0.00023 -0.00352 -
0.02188 -0.03823 0.37132 0.09455 -0.00001 0.09812 -0.00001 -
0.02849 -0.02788 -0.02831 -0.03360 -0.32018 0.00089 -0.02716 -
0.00001 -0.01593 -0.09840 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -
0.00009
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 -0.10068 -0.19181 0.51394 -0.26623 0.0 0.0
0.0 0.0 0.0 0.41046
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 -0.00554 0.40256 0.28245 -0.02082 0.0 0.0
0.0 0.0 0.0 0.30346
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.41660 -0.18372 -0.45267 -0.27504 0.0 0.0
0.0 0.0 0.0 0.56517
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.34446 0.14023 0.16984 0.49796 0.0 0.0
0.0 0.0 0.0 0.18802
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 -0.45464 -0.22829 -0.40655 0.05537 0.0 0.0
0.0 0.0 0.0 0.83529
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 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

D.4 10-digit Classification Problem

Network configuration and syaptic weights for the input layer are omitted here. Only those for the hidden layer and output layers are listed.

- Network configuration when the DMNN is trained with ideal digits:

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

- Test patterns:

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

- Synaptic weights when the DMNN is trained with ideal digits

-0.09752 0.05734 0.06594 0.02504 -0.07575 0.08018 0.03322
0.10652 0.04594 0.05742 0.18476 0.05972 0.13137 0.04851
0.05741 -0.00674 0.10564 0.06765 0.04245 -0.14875 0.08951 -
0.02896 0.09750 0.12626 0.12033 -0.03485 0.06296 0.13409
0.12016 0.16164 -0.13697 0.13909 0.10463 0.16412 -0.23052 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.11706
0.11932 0.01467 0.01468 0.01890 0.05029 0.03684 0.04421
0.04226 0.10216 0.00382 0.00964 0.00112 0.08886 0.11129
0.00386 0.11369 0.15921 0.22339 0.23295 0.00818 -0.03141
0.10409 0.07540 0.12053 0.0 -0.00087 0.03924 0.05664 0.06075
0.00001 0.12389 0.02585 -0.00577 -0.04304 0.12098 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.00473
-0.00572 0.0 0.0 0.06093 -0.06116 0.02555 0.06946 -0.01539
0.13387 0.08775 -0.0 0.06283 -0.02711 0.19250 0.17697 0.24441
-0.09391 -0.00746 0.00198 0.08359 -0.07443 0.21739 -0.01518
0.13108 -0.00903 0.03705 0.00334 -0.03828 0.17885 -0.0 -
0.02842 0.01111 0.01110 0.11307 -0.02298 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.04669
0.05018 0.00464 0.01342 0.00001 0.20902 0.09492 0.02156
0.02294 0.05640 -0.01205 0.15943 0.02824 0.08108 -0.02251 -
0.01663 0.29761 0.03083 0.03090 0.02278 0.03117 0.05701
0.04185 0.07798 0.04331 0.08094 0.03477 0.00701 0.01256
0.00169 0.06303 0.05647 0.04906 0.01583 0.03192 0.05382 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.03554
-0.01085 0.05525 0.07678 0.13699 0.00498 0.00417 0.02969 -
0.01228 -0.00271 0.24790 -0.0 0.02028 -0.01327 -0.00928
0.19571 0.12174 0.04108 0.06943 0.05489 -0.01319 0.00004
0.11020 0.05245 -0.02145 0.04980 0.02003 0.02641 0.00919
0.00448 0.04591 0.02156 0.09751 0.10619 -0.03577 0.00345 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
-0.0
-0.00649 0.00001 0.00001 0.0 -0.0 -0.0 0.06749 0.11148 0.04715
0.16830 0.00311 0.03780 0.15265 0.08140 0.19592 0.09016
0.13864 0.12094 0.06774 0.08726 0.07073 -0.07656 0.17345
0.03901 0.02104 -0.11764 0.01648 0.20297 0.07300 0.01977 -
0.09348 -0.02916 -0.06497 -0.10796 0.06959 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
-0.05497 0.04991 0.05749 0.00565 -0.06612 0.02742 0.09288
0.19228 0.00024 0.03460 0.00004 0.02465 0.20928 0.06725
0.01666 0.25984 -0.01177 0.04629 0.00931 0.24976 0.01842
0.00094 0.11740 -0.00768 -0.0 0.01229 0.01319 0.16132 0.00240
0.0 0.15769 0.10975 0.07778 0.00347 0.13678 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.00044
-0.09534 0.10355 0.05481 0.06735 0.03695 0.28476 0.05467
0.07675 0.05512 0.10979 0.09053 0.00743 0.08243 0.04084
0.14989 0.06032 0.00940 0.09850 0.03113 -0.17284 -0.02738

0.17722 0.08684 0.04199 -0.04357 0.08628 0.01343 0.03651
0.00658 -0.12145 -0.08413 -0.02417 -0.02417 0.00848 0.17315
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.07090
0.05218 0.14246 0.10362 0.09479 0.12942 0.11250 0.09547
0.11746 0.01009 0.01969 0.07223 0.01281 0.03005 -0.05861
0.06671 0.22517 0.03058 0.05785 0.03502 0.0 -0.31113 0.12163
0.09563 -0.02005 0.09230 -0.18255 0.03726 0.07286 0.05584 0.0
0.04419 0.08888 0.05562 0.08416 -0.00661 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.09989
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.16514 -0.24909 0.28422 0.13901 0.08368 0.02014 0.31402 -
0.03160 0.08577 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.17076
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.11571 -0.04993 -0.19950 -0.08235 -0.15149 0.13378 0.33791
0.20214 0.08315 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.34950
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -
0.16015 0.12344 0.28576 -0.13181 0.17922 -0.22724 0.00501
0.40184 0.05519 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.29601
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.10530 0.21357 0.06460 -0.30871 0.27534 0.04419 0.24676 -
0.39926 0.12280 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.42353
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.08172 0.09286 0.31585 -0.02871 -0.27311 0.10249 -0.06888 -
0.03416 -0.01986 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.46186
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.22714 0.00539 0.05193 0.25820 -0.00550 -0.33607 -0.08048
0.06213 0.42515 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.17671
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -
0.20254 0.35705 -0.16883 0.39576 0.15443 -0.08885 0.21054 -
0.10321 -0.31080 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.47084
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -
0.25598 -0.25588 -0.14726 0.10594 -0.12068 0.15007 -0.33021
0.03323 0.14128 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.82927

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.37505 0.16085 -0.20741 0.00547 0.12756 0.23209 -0.22344
0.20175 -0.20136 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.37534
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -
0.0 0.06200 0.07653 0.14613 0.17564 0.37514 0.09380 0.13828
0.10297 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 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

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] B. Widrow, "Generalization and Information Storage in Networks of ADALINE Neurons," in Self-Organizing Systems, Spartan Books, Washington DC, 1962.

[2] M. Minskey and S. Papert, Perceptrons, MIT Press, Cambridge MA, 1969.

[3] S. Grossberg, Studies of Mind and Brain: Neural Principles of Learning, Perceptron, Development, Cognition, and Motor Control, Reidel Press, Boston, 1982.

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

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

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

[7] T. Kohonen, Self-Organization and Associative Memory, Springer-Verlag, Third Edition, Berlin, 1988.

[8] 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.

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

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

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

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

[13] N.H. Farhat, D. Psaltis, and E. Paek, "Optical Implementation of the Hopfield Model," Applied Optics, Vol. 24, pp. 1469-1475, 1985.

[14] J.P. Sage and R.S. Withers, "An Artificial Neural Network Intergrated Circuit Based upon MNOS/CCD Principles, " American Inst. of Physics, Neural Networks for Computing, pp. 381-385, 1986.

[15] H.P. Graf, W. Hubbard, L.D. Jackel, and P. deVegvar, "A CMOS Associative Memory with Several Hundreds of Neurons," American Inst. of Physics, Snowbird, Utah, pp. 182-187, 1986.

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

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

[18] W. Wike, D.V. derBout and T. Miller III, "The VLSI Implementation of STONN," Proc. IJCNN, Vol. III, pp. 529-534, 1990.

[19] S. Garth, "A Chipset for High Speed Simulation of Neural Network Systems," Proc. IEEE ICNN, Vol. III, pp. 443-452, 1987.

[20] J. Rasure, D. Hush, J. Salas, and M. Newell, "A VLSI Three Layer Artificial Neural Network for Binary Image Classification," Proc. Int. Neural Network Society (poster session), 1988.

[21] S. Kung and J. Hwang, "Parallel Architecture of Artificial Neural Network," Proc. IJCNN, Vol. II, pp. 165-172, 1988.

[22] C.E. Atlas and Y. Suzuki, "Digital Systems for Artificial Neural Networks," IEEE Circuits and Devices, pp. 20-24, Nov. 1989.

[23] Y. Hirai, K. Kamada, M. Yamada, M. Ooyama, "A Digital Neurochip with Unlimited Connectivity for Large Scale Neural Networks," Proc. IJCNN, Vol. II, pp. 163-169, 1989.

[24] 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.

[25] S. Gilbert, "A Generic Architecture for Wafer-Scale Neuromorphic Systems," Proc. IJCNN, Vol. III, pp. 501-513, 1989.

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

[27] P. Miller, J.V. derSpiegel, D. Blackman, T. Chiu, T. Clare, J. Dao, "A General Purpose Analog Neural Computer," Proc. IJCNN, Vol. II, pp. 177-182, 1989.

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

[29] D.F. Specht, "Probabilistic Neural Networks for Classification, Mapping, or Associative Memory," Proc. IJCNN, Vol. I, pp. 525-532, 1988.

[30] A.F. Murray and A.V.W. Smith, "Asynchronous VLSI Neural Networks using Pulse-stream Arithmetic," IEEE J. of Solid-State Circuits, Vol. 23, no. 3, pp. 688-697, June 1988.

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

[32] 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.

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

[34] F.M. Salam, "A Tutorial Workshop on Neural Nets and Their Engineering Implementation," 31st Midwest Symp. on Circuits and Systems, St. Louis, Aug. 1988.

[35] L.O. Chue and G.N. Lin, "Nonlinear Programming without Computation," IEEE Trans. on Circuits and Systems, Vol.31, pp. 182-188, Feb. 1984.

[36] M.P. Kennedy and L.O. Chua,"Unifying the Tank and Hopfield Linear Programming Circuit and the Canonical Nonlinear Programming Circuit of Chua and Lin," IEEE Trans. on Circuits and Systems, Vol. 34, pp. 210-214, Feb. 1987.

[37] W.E. Weideman, "A Comparison of a Nearest Neighbor Classifier and a Neural Network for Numerical Handprint Character Recognition," Proc. IJCNN, Vol. I, pp. 117-120, 1988.

[38] T. Irino and H. Kawahara, "A Method for Designing Neural Networks Using Nonlinear Multivariate Analysis: Application to Speaker-Independent Vowel Recognition," Neural Computation, Vol. 2, pp.368-397, 1990.

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

[40] S. Hosogi, "Manipulator Control Using Layered Neural Network Model with Self-Organizing Mechanism," Proc. IJCNN, Vol. II, pp. 217-220, 1990.

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

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

[43] J. Hertz, A. Krogh, and R.G. Palmer, Introduction to the Theory of Neural Computation, Addison Wesley, pp. 89-111, 1991.

[44] M. Yu, "A Study of the Applicability of Hopfield Decision Neural Nets to VLSI CAD," 26th ACM/IEEE DAC, pp. 412-417, 1989.

[45] S.E. Fahlman, "Fast-Learning Variations on Back-Propagation: An Empirical Study," Proc. Connectionist Models Summer School, Pittsburgh, pp. 38-51, 1988.

[46] D. Plaut, S. Nowlan, and G. Hinton, "Experiments on Learning by Back Propagation," Technical Report CMU-CS-86-126, Dep. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1986.

[47] T.P. Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, "Accelerating the Convergence of the Back-Propagation Method," Biological Cybernetics 59, pp. 257-263, 1988.

[48] R.P. Gormann and T.J. Sejnowski, "Learned Classification of Sonar Targets Using a Massively-Parallel network," IEEE Trans. on Acoustics, Speech, and Signal Processing, pp. 1135-1140, 1988.

[49] Y. Le Cun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," Neural Computation, pp. 541-551, 1989.

[50] Y. Le Cun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.D. Jackel, "Handwritten Digit Recognition with a Back-Propagation Network," Advances in Neural Information Processing Systems, Vol. 2, pp. 396-404, 1989.

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

[52] G.E. Hinton and T.J. Sejnowski, "Learning and Relearning in Boltzmann Machines," Parallel Distributed Processing, Vol. 1, Chap. 7, Cambridge, MIT Press, 1986.

[53] K. Gutzmann, "Combinatorial Optimization Using a Continuous State Boltzmann Machine," Proc. IEEE ICNN, Vol. III, pp. 721-734, 1987.

[54] S. Eberhardt, T. Duong, and A. Thakoor, "Design of Parallel Hardware Neural Systems from Custom Analog VLSI 'Building Block' Chips," Proc. lJCNN, Vol II, pp. 545-556, 1990.

[55] D.B. Schwartz and R.E. Howard, "A Programmable Analog Neural Network Chip," Proc. IEEE Custom Integrated Circuits Conf., IEEE Cat. No.:88CH2584-1, pp. 10.2.1-10.2.4, 1988.

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

[57] J. Raffel, J. Mann, R. Berger, A. Soares, and S. Gilbert, "A Generic Architecture for Wafer-Scale Neuromorphic Systems," Proc. IEEE ICNN, Vol. IV, pp. 485-493, 1987.

[58] R. Hecht-Nielsen, Neurocomputing, Addison-Wesley, pp. 272-297, 1990.

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

[60] M. Holler, S. Tam, H. Castro, and R. Benson, "An Electrically Trainable Artificial Neural Network (ETANN) with 10240 'Floating Gate' Synapses," Proc. IJCNN, Vol. II, pp. 191-196, 1989.

[61] 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.

[62] 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.

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

[64] 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.

[65] 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, Glasgow, May 1989.

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

[67] S.W. Golomb, Shift Register Sequences, revised ed., Laguna Hills, CA, Aegean Park Press, 1982.

[68] A.K. Jain, "Pattern Recognition," Intl. Ency. of Robotics: Application and Automation, 1988.

[69] A Design & Test, "Behavioral Description Languages," IEEE Design & Test of Computer, pp. 56-68, 1990.

[70] J.R. Armstrong, "Chip-level Modeling with HDLs," IEEE Design & Test of Computer, pp. 8-18, 1988.

[71] C.J. Tseng, R.S Wei, S.G. Rothweiler, M.M. Tong, A.K.Bose, "Bridge: A Behavioral Synthesis System for VLSI," IEEE CICC, pp. 2.6.1-2.6.4, 1988.

[72] G. Borriello and E. Detjens, "High-level Synthesis; Current Status and Future Directions," 25th ACM/IEEE DAC, pp. 477-482, 1988.

[73] M. Shahdad, R. Lipsett, E. Marschner, K. Sheehan, and H. Cohen, R. Waxman, D. Ackley, "VHSIC Hardware Description Language," IEEE Computer, pp. 94-103, Feb. 1985.

[74] IEEE Standard VHDL Language Reference Manual, IEEE std. 1076-1987, IEEE Inc., Mar. 31, 1988.

[75] VHDL User's Manual, Volume 1: Tutorial, Intermetrics Inc., April 1987.

[76] R. Stansie, M. Brown, "VHDL Modeling for Analog Digital Hardware Design," ICCAD, 1989.

[77] R. Stansie, M. Brown, "Using VHDL as a Language for Describing the Behavior of Analog and Mixed Analog/Digital Systems," VHDL Users Group, April 1990.

[78] Y. Xie and M.A. Jabri, "Analysis of the Effects of Quantization in Multilayer Neural Networks Using a Statistical Model," IEEE Trans. on Neural Networks, Vol. 3, No. 2, pp. 334-338, Mar. 1992.

[79] C.Y. Maa and M.A. Shanblatt, "Linear and Quadratic Programming Neural Network Analysis," IEEE Trans. on Neural Networks, Vol. 3, No. 4, pp. 580-594, July 1992.

[80] J.V. Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Reliable Components," in Automata Studies, Prinston, N.J., Princeton University Press, pp. 43, 1956.

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

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

[83] P. Gupta and R. Kumaresan, " Stochastic and Deterministic Computing with Pseudo Random Sequence: Application to Digital Filtering," 12th Annual Conf. on Signals, System and Computers, Pacific Grove, CA, pp. 159-163, Nov. 1986.

[84] B.W. Lindren, Statistical Theory, NY, Macmillan Company, pp. 59-75, 1962.

[85] Y.H. Pao. Adaptive Pattern Recognition and Neural Networks, Addison Wesley, pp. 3-21, 1989.