VLSI Implementation of a Digital Neural Network with On-Chip Learning

by

Darrell Gates

A thesis
presented to the University of Manitoba
in partial fulfillment of the requirements
for the degree of

Master of Science

Department of
Electrical and Computer Engineering
University of Manitoba
Winnipeg, Canada 1990

ISBN 0-315-71831-5

VSLI IMPLEMENTATION OF A DIGITAL NEURAL NETWORK WITH ON-CHIP LEARNING

by

Darrell Gates

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

This thesis explores the VLSI implementation of a neural network with on-chip learning capabilities. This digital ASIC is intended to be used as the basis for a high-speed neural accelerator board. A unique digital neural network architecture is presented and analyzed. A novel VLSI design style is introduced and developed. Implementation details and design issues are also presented. Fundamental issues of neural networks are introduced and major learning paradigms are discussed. A thorough examination of digital ANNs is presented.

Acknowledgements

With the encouragement and support of Dr. Howard Card, this thesis has finally become a reality. Many thanks to the 'boys' in the VLSI lab.
A special thanks to DB.

This work was partially supported through funding from the Natural Sciences and Engineering Research Council of Canada and through equipment loans from the Canadian Microelectronics Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Intelligence (AI) is a broad term which encompasses all computer systems that can exhibit some form of human intelligence. These *human-like* computers are capable of performing operations that are analogous to the human capacities of learning and decision-making.

Computers that can exhibit some form of human-like intelligence based upon certain rules or statistical inferences have been the mainstream of traditional AI expert systems. These expert systems have traditionally been of assistance in situations involving the laborious task of making sense out of large amounts of data (such as the databases currently being used in the medical profession for making preliminary diagnoses) or in performing tasks that any human would consider to be too tedious (such as a robotic arm used in manufacturing) or too dangerous (such as a robotic arm used in a nuclear energy plant). However, all of these traditional AI systems suffer from serious flaws. These expert systems require permanent operators and preprogrammed sets of rules for their operation—all of which have a dramatic effect on the overall ability of the system to perform well in new environments and under different circumstances.

A relatively new area of AI has emerged in the last few years. Researchers, in order to distance themselves from traditional AI and to emphasize the fact that the field contains new approaches to AI, have coined the term **neural networks** for this *new* discipline.

In the 1950s, researchers discovered that architectures of simple processing elements configured in a specific order and performing simple calculations could exhibit brain-like properties. That is, a network of interconnected processing elements[1] could make semi-intelligent decisions based upon experience alone, and could easily adapt to new situations without having to be reprogrammed. This discovery was in sharp contrast to traditional AI theories, wherein large rule-based systems that had been painstakingly programmed with responses to a certain situation could be outperformed by a simple neural network that had been **trained** to respond to the same situation. A neural network, like a human, could be taught to recognize and recall without the use of any a priori knowledge of a particular problem—only a training set of typical input/output data was required.

These two branches of AI, i.e. neural networks and traditional AI, were now competing for research supremacy. In the 1960s, a book on *Perceptrons* [1] dealt a harsh blow to the field of neural networks. The basic premise of this book was that a neural network could **not** perform even the simplest of tasks, such as the XOR problem,[2] and that the expert systems developed by the traditional AI theorists were the **only**

[1]These interconnected processing elements were called 'neurons' and 'synapses' to stress the fact that the algorithms being implemented came from brain theories in psychology and biology.

[2]The XOR problem can be stated as follows: The XOR (exclusive OR) of two inputs is TRUE if either (but not both) of the inputs are TRUE.

systems that could mimic true human-like behavior. For almost the next two decades, most AI research would be guided by traditional rule-based expert systems. Neural network researchers (connectionists) were relegated to the backroom—it appeared as though neural networks had experienced their demise.

However, true and dedicated connectionists were quietly working in their *underground* laboratories applying discoveries that biologists and psychologists had made about the brain. The neural network revolution was slowly maturing; and, with the publication of *Parallel Distributed Processing* [2, 3] in the eighties, there was a resurgence of interest in neural networks.

Connectionists were stressing the fact that neural networks possessed 'natural intelligence' as opposed to 'artificial intelligence' — that neural networks contained implementations of the *natural* way in which the brain functions. The notion of a computer that could mimic the human mind, and actually think, had researchers from all disciplines jumping on the proverbial 'bandwagon'. Imagine—a computer that could actually think, reason, make judgements, and make the same decisions that we, as humans, might also make! This excitement was also fueled by the multitude of science fiction writers who envisioned worlds run by computers, and Hollywood movie moguls who instilled in us the visions of futuristic computer technology with human-like qualities—little did they know that they were just 'fueling the fire' for connectionists to showcase their novel ideas and present an enhanced discipline that seemed far superior to traditional AI. Neural networks was once again being considered as a credible research field.

Traditional AI systems have failed when it comes to being able to make intelligent choices in unanticipated situations; this is where neural networks can excel and should be used to take over from where a traditional AI system leaves off.

Picture the following scenarios:

> . . . You are commissioned with the responsibility for the safety of all passengers on a major commercial airline. You are aware that the latest threat to airlines is plastic explosives concealed in luggage and you know that an x-ray machine to discover weapons being smuggled aboard has been 'pushed to the limit'. A trained dog could be used to sniff all baggage before it is loaded onto the plane; however, most passengers would not appreciate the delays and a trained nose of a dog could be tricked with other overpowering scents. You, yourself, could examine every piece of luggage that is checked, but this would be a very slow and expensive process. Imagine if you could employ a computer (something like a robotic dog) that was trained to 'sniff' every piece of luggage and make an intelligent decision as to the possibility of a concealed bomb, without the use of any type of human operator (which an x-ray machine requires).
>
> . . . You are driving to the restaurant and you notice that your dinner reservations are not for another hour. You pick up the phone and advise the limousine driver to drive around the city so that you can impress the lady that you are with and get to know her a little better. The car speeds up and takes you past all the hot spots—so that you can impress your friends too. You finally arrive at the restaurant, on time, and you're a big hit with the lady. Out of the kindness of your heart you decide to tip the limousine driver $50 before you go. As you lean into the front seat, you discover, to your amazement, that the driver is actually a computer that has been trained to drive the car (and all along you thought he was just short). You can't believe the skill with which this machine has negotiated all the hazards of the road. Since you are an educated engineer, you marvel at the reality of such a futuristic machine, pocket the $50, and enter the restaurant.
>
> . . . You are engaged in some type of intelligent conversation with a machine that makes human-like decisions and responds through verbal communication, and accepts and executes verbal orders in a quasi-intelligent manner. Imagine the possibilities . . . a computer that could actually think and talk.

Does this sound like science fiction? Or does it touch on reality? All of these *stories* were once in the minds of people with 'wild' imaginations; however, with current advancements in neural network technology, these futuristic idealities have almost become a present-day reality.

In fact, a system that can detect bombs concealed in baggage is already operating at several major international airports in the world. This system, called **SNOOPE** (System for Nuclear On-line Observation of Potential Explosives), developed by Science Applications International Corp. (SAIC), uses a neural network that makes decisions about baggages that may contain a bomb. SNOOPE is mainly hardware-based with an expert system and a neural network that complement each other in the decision-making process. The neural network takes over from the point where the rule-based system tends to fail. According to an SAIC sales brochure, the system works by exposing the luggage to "low energy (thermal) neutrons and analyzing the gamma rays resulting from neutron absorption by atomic elements in the luggage. The computer software searches for specific combinations of atomic elements that characterize explosives." The expert system follows rules that are preprogrammed for known explosive compositions while the neural network has been trained to recognize these specific compositions. The neural network can also generalize to other combinations of atomic elements that may characterize explosives—compositions that have not been specified in the expert system. The system does not require a human operator to interpret the data; rather, the system makes an intelligent decision based on its current knowledge.

Real-world problems require neural networks that can generalize in situations not previously encountered. Being able to properly generalize is important for an unmanned vehicle to navigate a territory that is unfamiliar. This is exactly the case with **ALVINN** (**A**utonomous **L**and **V**ehicle **In** a **N**eural **N**etwork), developed at Carnegie-Mellon University (CMU) by D.A. Pomerleau et al., where the goal is to allow the vehicle to navigate along a winding road. Sensor inputs to the neural network are through video cameras that act as a "video retina" [4]. The neural network embedded in the computer system of the vehicle is trained to recognize certain features of a typical roadway, such as trees and road edges, and is released on a previously uncharted roadway. The neural network has also been trained to steer the vehicle and to accelerate and decelerate. In order for the vehicle to make a completely autonomous navigation of the roadway, the training set is very important. If not enough information has been provided during training, the vehicle may not respond correctly. If it is given too much information, the vehicle may not be able to adapt to new situations. Presently, the vehicle can accurately drive at $3\frac{1}{2}$ mph along a short path, under various conditions of weather and lighting. This is double the speed of that achieved by ALVINN using a non-neural network system in attempting to drive the same pathway. While 'unmanned driving' has not yet progressed to the same level of confidence as that of the limousine driver pictured in our scenario, it is definitely a step in the same direction.

One of the classic examples of a neural network that demonstrates impressive potential is NETtalk; a computer program developed by Terence Sejnowski and Charles Rosenberg. The program configures a neural network that learns to read English text aloud without having to use linguistic rules. The neural network is shown a textual conversation that had been transcribed by a linguist into the correct phonemes. The difference between the network's response and the correct response is 'how' the neural network is trained to read the text. Just like children who learn to speak by listening, trying and being corrected, the neural network will 'pick up on' all the regularities and exceptions in the pronunciation of the English text. This flexible learning process is in stark contrast to the rigid requirements of an expert system, which must have a large database of linguistic rules, words and pronunciation examples stored in memory. The English language is very confusing—we don't always follow the rules or say words as they are noted in the dictionary. It would take an expert system just to program all the rules into another expert system that could handle English speech as well as NETtalk does. However, neural networks, in this extraordinary example of their power, have gotten us a step closer to the talking computer.

The three systems presented are excellent examples of neural networks currently operating in the 'real world'. Granted, they exhibit only some resemblance to the scenarios pictured previously, but the use of neural networks is currently in its infancy. However, the fine line between science fiction and reality is becoming thinner with each new advancement in neural networks.

Although these systems exhibit three very different approaches to neural network implementations, they all utilize the same learning paradigm. A learning paradigm is a rule (or law or algorithm) that a neural network uses to modify its internal structure so that it may exhibit learning characteristics. There are many neural network architectures and many associated learning paradigms that are optimized for certain applications. One of these paradigms is the backpropagation of error learning rule (sometimes referred to as backpropagation, or simply backprop) that is used in each of the above examples to train the network to respond in a human-like manner. Both SNOOPE and ALVINN use dedicated hardware to implement the various equations that are inherent in the backpropagation algorithm. This algorithm requires many multiplications and additions that are performed in parallel—current Von Neumann-style serial computers do not have the power to efficiently execute this algorithm. Therefore, SAIC and the researchers at CMU have each developed a general purpose digital signal processing hardware board that is used to accelerate most of the calculations required by the backpropagation algorithm. These systems operate mainly in a sequential fashion, and are optimized to do multiplication and addition in much the same way as today's Digital Signal Processing (DSP) integrated circuits. Thus, these systems are deficient of the inherent parallelism which gives neural networks their strength. Software could be partitioned into a parallel structure and run on a parallel computer, such as the Connection Machine; however, parallel computers are not that accessible.

Although these implementation strategies offer flexibility and short-term adequacy, we feel that the ultimate solution to making a computer that mimics the human mind is the implementation of a neural network in silicon. The advancement of Very Large Scale Integration (VLSI) technology has made the implementation of neural network paradigms[3] in a Application Specific Integrated Circuit (ASIC) a reality.

1.1 Purpose

The purpose of this thesis is the direct implementation of a neural network that is capable of performing on-chip learning utilizing the Backpropagation Learning Algorithm in a fully-digital VLSI ASIC. This chip is intended to be used as the basis for a neural hardware accelerator board for the purpose of increasing simulation speeds of feedforward neural network models.

[3]A neural network paradigm refers to a neural network architecture and its underlying learning paradigm.

Since the current trend in artificial neural network research focuses on analog devices, we also want to look at the role that electronic digital hardware will play in future implementations of neural network models.

1.2 The Problem

Artificial Neural Networks (ANNs) are still in their infancy and until parallel computing architectures become the *norm*, connectionists will have to simulate neural network paradigms on traditional Von Neumann-style serial computers that are conventionally slow for massively parallel problems such as neural networks. The alternative, in the meantime, is to build electronic architectures that *emulate* neuronal behavior with the hope that these structures will form the basis for a parallel computing architecture based on neural networks.

DSP based hardware accelerator boards are the current 'hot topic' of digital neural network designs. These ANNs rely on the DSP to perform learning algorithms in essentially a sequential manner. Neural networks are inherently parallel by nature and attribute their speed to this massive parallelism. Designing parallel architectures using DSPs is not very practical; costs of optimized signal processing chips can be horrendous. Silicon implementations of neural networks can be very cost effective in modelling massive parallelism. However, most neural network ASICs do not include the learning algorithm on-chip; they usually depend on a host computer that is typically serial.

Learning paradigms can be very slow on traditional serial computers. Performing the backpropagation learning algorithm for large networks on a serial computer is just not practical; the learning time increases tremendously with the number of neurons, the problem size, and the training set. The short-term alternative is to implement the algorithm in silicon as an integrated circuit. This will relieve the host computer of the tedious task of arithmetic calculations. If we can optimize hardware to perform this, or any other, learning paradigm, we can obtain results much faster and increase productivity.

The necessity for an ASIC that will perform a neural network paradigm is to create circuits that are direct implementations of biological neurons that will eventually be used to form a human-like *brain* structure; DSP accelerator boards are nothing more than enhancements to current computing power and will eventually become engulfed by advancements in computer technology. A VLSI implementation will allow the neural network researcher to optimize the algorithm being implemented.

1.3 Scope

We begin this thesis with a study of the background of neural networks. A brief history of neural network development is presented — including some major events that shaped the field. A typical neural network model is described and various typologies of neural networks are reviewed. We state some of the more interesting learning paradigms with a major emphasis on Backpropagation learning. We conclude the Background chapter with a thorough examination of ANN implementations.

The next chapter describes a VLSI implementation of the Backpropagation algorithm. We begin this chapter with a discussion on implementation issues regarding the design of the major components within the neural network structure. A section on a novel VLSI design technique is presented. We also present our implementation of the algorithm in a VLSI ASIC — describing the major components within the design and how they are linked to form the network.

Finally, we present our conclusions and recommendations for further study.

Appendix A offers a mathematical derivation of the Backpropagation algorithm from its roots in the Perceptron Convergence Theory to its current generalization of the Delta Rule.

Chapter 2

Background

2.1 Neural Networks

Neural networks can be described as an attempt to create a machine that mimics the thought processes of the human mind; a machine that can, by intuition and inference, make sense out of incomplete, confusing or *fuzzy* information. Neural networks exhibit characteristics of biological models of brain-like behavior. *An artificial human brain?* Well, maybe. But current neural network models come no where near the complexity of the human brain. **Table 2.1** shows the relationship between biological processes in the brain and current digital computer architectures of neural network models. The human brain contains about 100 billion neurons, each of which connects to about 10,000 other neurons in a massively parallel structure. The brain can perform most cognitive processes in less than 500 ms even though the neuron's speed has been estimated at around only 1 ms, simply by this inherent parallelism. A human can perform *simple* vision tasks that thousands of super computers can't. But a simple pocket calculator can outperform a human 10-to-1 on explicit arithmetic problems. As you can see, brains and computers do not compute the same.

ELEMENT	BRAIN	NEURAL NETWORK
Organization	Network of Neurons	Network of Processing Elements (PEs)
Architecture	10^{11} neurons	$<$ 100 PEs
Hardware	Neurons	Switching device (transistor)
Structure	10^4 connections/neuron	$<$ 100 connections/PE
Technology	Biological	Silicon
Processing	Analog	Digital or analog
Speed	10^{-3} sec	10^{-9} sec

Table 2.1: **Comparison of the Brain and Neural Network Computing** [20]

Neural networks spread the problem into smaller tasks and distribute the processing to its simple parallel processors. The brain uses this *parallel distributed processing* to perform cognitive tasks such as natural language understanding, abstract reasoning, concept generalization, knowledge processing, pattern classification, computer vision, association and evaluation. Conventional computers are outstanding at arithmetic processing, sorting and logic; all processes that the human brain has the potential to do, but cannot do with any efficiency.

Neural networks are simple systems that are capable of complex behavior. By simplifying the processes of biological neurons, a system that *reasons* can be developed. This development is an indirect reaction to the failure of traditional artificial intelligence (AI) research into the laws and rules governing cognition. Decades of AI research has not resulted in substantial progress, hence the rebirth of neural networks and *natural intelligence* (NI).

2.1.1 Chronology of Events

Modern neural network theories can be traced back to ideas first introduced in the 1940s and 1950s. In 1943, McCulloch and Pitts were developing mathematical theories on the nervous system and how neurons might work. [15] A few years later, Hebb introduced a *learning rule* whereby networks would reinforce response patterns that occurred most often, as though they were learning by experience. [16] This rule stated that a connecting synapse between two simultaneously active neurons should become 'stronger', while inactive neurons would cause a synapse to 'weaken'.

In the late 1950s, Rosenblatt developed networks that could learn to recognize simple patterns. [17] The *perceptron*, as it was called, could decide whether an input belonged to one of two classes. A single node would compute the weighted sum of binary-type inputs, subtract a threshold, and pass the result through a non-linear hard limiting threshold that classified the input. The perceptron formed a two decision region separated by a hyperplane. The *Perceptron Convergence Theory* was developed by Rosenblatt as a means for adjusting the weighted connections until a stable state was reached. Rosenblatt proved that if the inputs form two separable classes, then the output of the network will converge to one of these two classes. At around the same time, Widrow and Hoff were developing a theory to reduce the error between the actual output and the desired output of a perceptron-like network, in a least mean squared (LMS) sense. [18] The LMS algorithm is similar to the Perceptron Convergence Theory except for the fact that Widrow and Hoff's LMS algorithm uses a threshold-logic nonlinearity for the transfer function, so that more than two classes of output are formed. These two theories have formed the basis for most learning

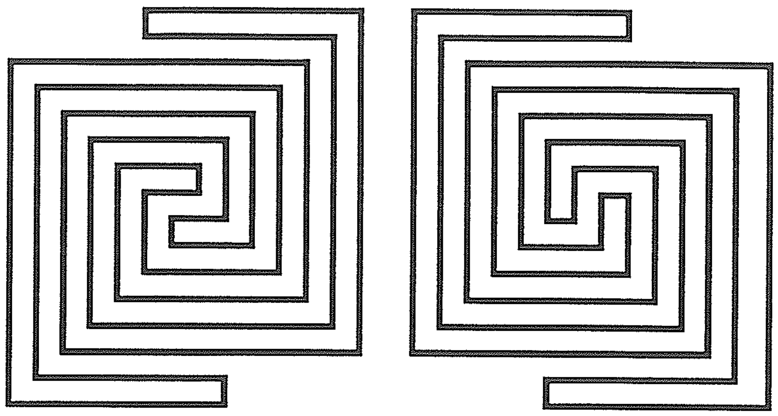


Figure 2.1: **Difficult Perceptron Convergence Task** [1]

algorithms to date. Recently, though, modifications to the Hebbian rule have been successful in the formation of other new learning algorithms.

In what mainstream connectionists might call a catastrophic event, Minsky and Papert 'slashed' the Perceptron Convergence Theory in their book on perceptrons [1]. In fact, though, Minsky and Papert showed that only a 'small' class of perceptrons could not perform certain tasks in pattern recognition. It turns out that some of the tasks the perceptrons could not perform are also extremely difficult for human vision—like discerning which "one of the two spirals [on the cover of their book] is made with a single line... [and which] is actually two lines nested within each other," (shown in **Figure 2.1**)[1] [56]. Nevertheless, the 1960s and 1970s proved fatal for neural networks. While traditional AI theories of symbolic processing flourished, NI theories vanished into relative obscurity. True connectionists, however, were not discouraged. Pioneers in the field, such as Grossberg and Kohonen, were producing networks and ideas throughout the 1970s [19].

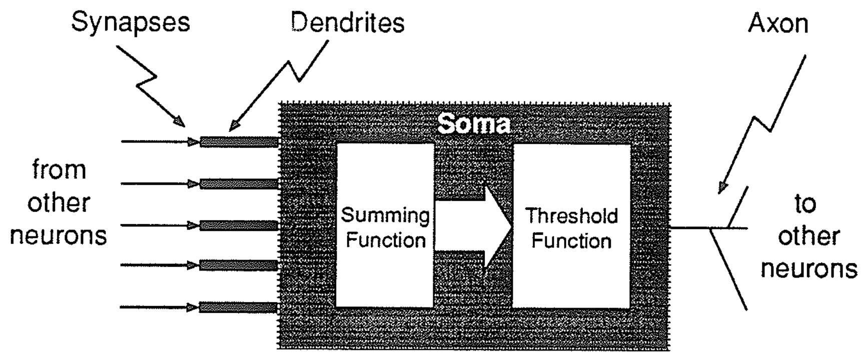
[1]It is easy for one to trace the lines by hand, but there are no visionary clues that tell us that the spiral on the right is a fully-connected single line.

In 1982, Hopfield discussed the application of analog non-linear interactions in neural networks and introduced ideas on electronic circuits for these networks [81]. It wasn't until 1986, when Rumelhart and McClelland introduced the concept of *Parallel Distributed Processing* (PDP) [2, 3], that neural networks regained their popularity. The PDP group of researchers introduced models based on nodes and connections, and the laws that changed the activities in them. The current flood of activity in neural networks is in relation to the concepts of subnetworks and connections analogous to the nature of brain parts. Engineers are currently learning and applying connectionist principles in the design of specific implementations of these subnetworks.

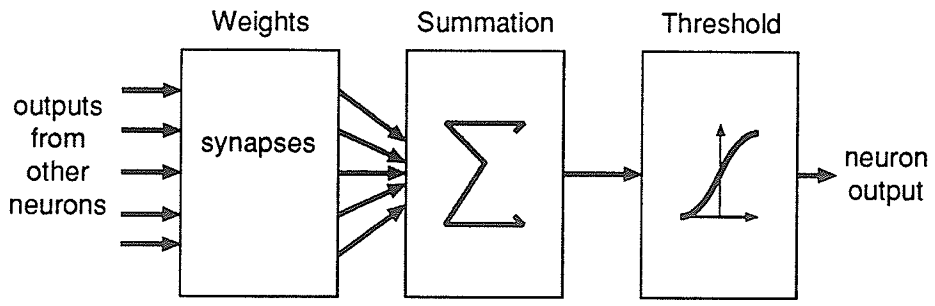
We have presented what we feel are the most influential events of the past five decades. Who knows what the next five decades might bring?

2.1.2 Model of a Neuron

It is important to understand how a biological neuron works in the human brain before attempting to design neural network models. **Figure 2.2(a)** is a simplified model of the structure of a neuron in the human brain. Basically, the neuron consists of a cell body (*soma*) whose physical shape seems to partially determine the function that it will perform in the brain. A complex set of interconnections add to this determination. The neuron's input connections (*dendrites*) accept electrical signals, that are either excitatory (positive) or inhibitory (negative), from other neurons' output connections (*axons*) through links, called *synapses*, that modify the axon's potential. It is theorized that the neuron performs a weighted sum of input signals and emits an output signal proportional to some threshold function. Recently, it has been suggested that computation also takes place outside the soma [20].



(a)



(b)

Figure 2.2: **Simplified Neuron Models** [20]: **(a) biological (b) artificial**

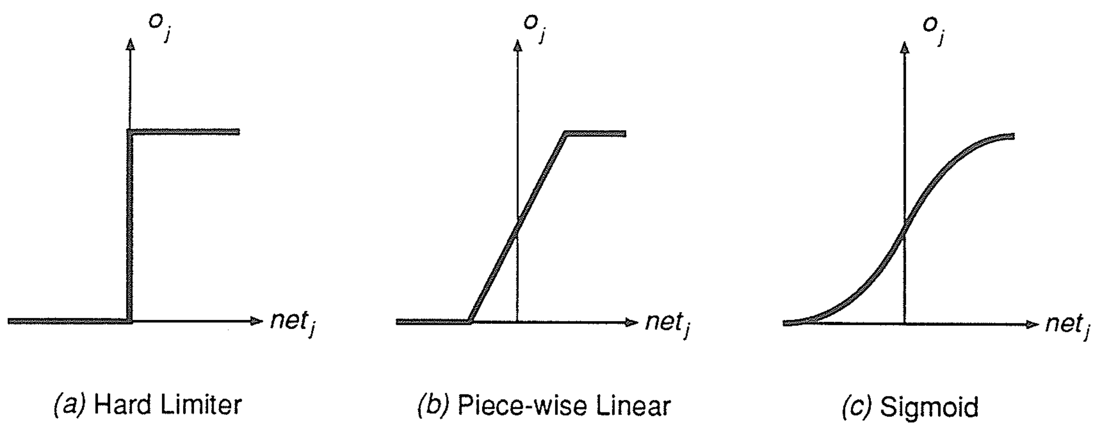


Figure 2.3: **Typical Activation Functions**

An artificial neuron, or processing element, is shown in **Figure 2.2(b)**. The simplified model consists of weighted input connections (analogous to synapses and dendrites), the summation and activation units (the soma), and the output connection (the axon). The input signals, usually positive, can become either excitatory or inhibitory, by using bipolar weights. The model performs a weighted summation of its inputs, passes this summation through a transfer function (the threshold) and produces an output activation that is sent to other neurons in a network. The transfer function defines the output state of a neuron. Typical transfer functions, of which the sigmoid is the most popular, are shown in **Figure 2.3**. In this simplified model of an artificial neuron, most of the biology is omitted. Models of neural networks are usually characterized by the topology of the network and its learning rules.

In the next few sections we will show some basic neural network topologies and state some important learning rules.

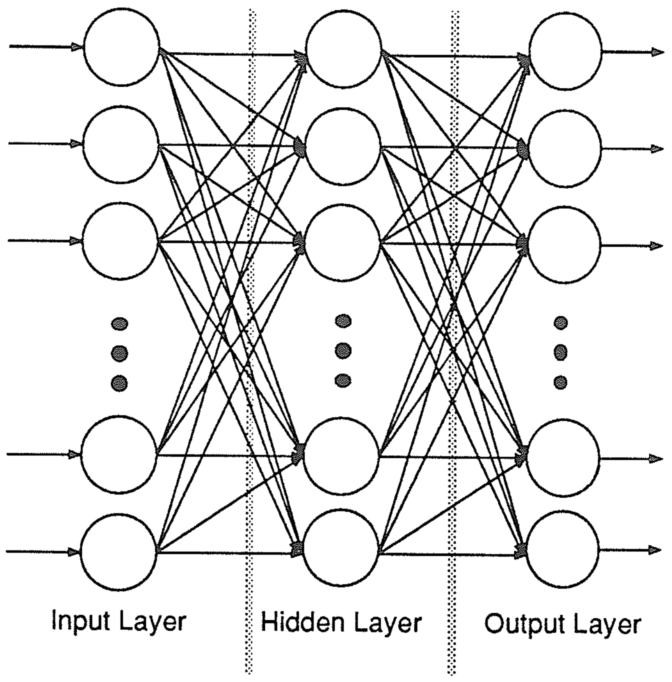


Figure 2.4: **Multi-layer Feedforward Network**

2.1.3 Topologies

Neural network behavior is usually characterized by the way in which neurons are interconnected. There are two major classes of neural network topologies.

The *feedforward* network usually consists of layers of neurons that are connected such that neurons in one layer can only connect to other neurons in a *higher* layer (connections to lower layers are forbidden). This type of network, usually called a *multi-layer feedforward network*, is shown in **Figure 2.4**. Input signals propagate forward through the network until they reach the output layer. Layers that are neither input nor output are referred to as *hidden* layers because they contain abstract representations of user-defined input/output values.

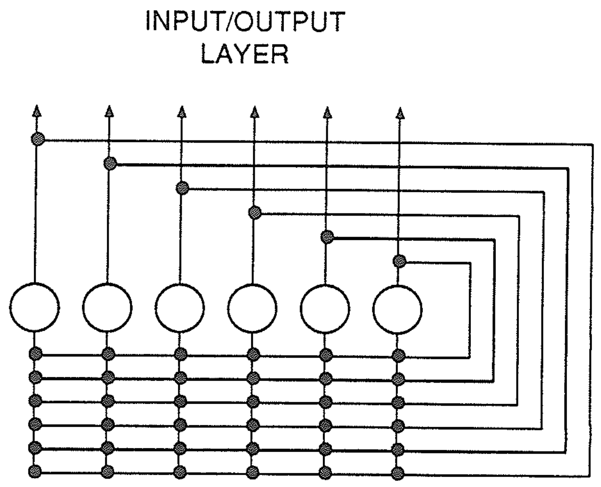


Figure 2.5: **Single-layer Feedback Network (Hopfield)**

The second class of topologies is the *feedback* network. Feedback networks can take many forms, such as a multi-layer feedforward network with the output layer connected to the input (sometimes called sequential networks) or neurons connected to each other on the same layer (usually called lateral inhibition or comparative networks). However, the most basic form of a feedback network (and one of the most useful) is the *Hopfield* neural network, shown in **Figure 2.5**.

The Hopfield network is used mainly as an associative memory, and is referred to as *self-organizing*. In this topology, all neurons are connected to each other in a grid-like structure and the layer of neurons functions as both an input and an output. It is interesting to note that although feedback networks are typically used for associative memories, there is no reason that a feedforward network could not also be used. In fact, it can be shown mathematically that a feedback network has an equivalent feedforward network that can perform an identical task [21].

A typical neural network model has been summarized in [2, 3]. The model includes processing units, states of activation, an output function, a network pattern of connectivity (topology), propagation rules for propagating patterns of activity through the network, activation rules for combining inputs and current state to create new activities, a learning rule for modification of connection strengths between units, and an environment in which the neural network can operate.

A neural network in its most general form looks like any other parallel processing architecture. In fact, the neural network model consists of processing elements (neurons), connections between processors (synapses) and information that passes passing along these connections. Just as a parallel processing architecture is defined by the topology of its interconnections, so too is the neural model.

2.2 Learning Paradigms

Without a program of instructions, a computer is a 'useless' machine. The program usually instructs the computer to perform specific tasks on a set of input data to create some sort of output. Therefore, the program is an essential part in a computer environment. We have seen how a typical neural network is constructed. If these types of networks are to be useful, we must be able to *program* them as well. Neural networks are not programmed in the conventional sense (it would be virtually impossible), they are *taught.*

Teaching a neural network cognitive knowledge is basically a modification of the synaptic weights according to some learning algorithm or rule. Therefore, we often say that the 'knowledge of a neural network is in the weights'. The analogy to human behavior modification is no coincidence, as neural networks try to mimic the human thought process. Learning rules are usually called *paradigms*, which simply means "an outstanding, clear and typical example, model or prototype" [22]. Learning paradigms are generally associated with a particular neural network structure. Just as there are many neural network topologies, there are equally as many learning paradigms. This section does not attempt to cover the myriad of approaches to teaching neural networks; rather, it is intended to introduce a few of the more interesting and popular learning paradigms, and to state the Backpropagation of Error learning paradigm or Generalized Delta Rule. Refer to **Appendix A** for a mathematical derivation of this learning paradigm.

Learning paradigms can be broken down into three classes [23]: (1) supervised learning, where a teacher provides an input and output training pair of vectors to the network; (2) reinforcement learning, which requires a single scalar evaluation of the output; and, (3) unsupervised learning, where only an input vector is supplied. Within each of these classes, learning takes place by modifying the synaptic weights after some type of error measure is performed.

In the case of supervised learning, an a priori knowledge of input/output responses is necessary (for example, the XOR logic problem), and the error between the actual response and the target response is used to *correct* the network. Reinforcement learning utilizes a global reward strategy whereby credit is assigned to a local decision by measuring the correlation between it and the global reinforcement signal (an example could be the 'Hot & Cold game' where someone is guided by 'hot' and 'cold' verbal signals). Unsupervised learning requires only that an input signal be presented to the network. After a set of trials are presented, a grade or performance measure is taken at the output and a *score* that tells how well the network is doing is created. Within each class there are various learning procedures (sub-classes) that are distinguished only by how they treat the error signal; as usual, there are some exceptions. Learning in a neural network occurs as the weights are changed to reduce the errors — in effect, the network gains experience [21]. The learning procedure is usually stopped after a set of training patterns are successfully *learned* by the network. The final pattern of synaptic weights represents the knowledge of the system and is generally fixed after training.[2]

Unsupervised learning discovers the underlying structure of variables and associates a correct response. A popular unsupervised learning procedure is the Kohonen self-organizing feature maps, as described in [24]. A two-dimensional array of output units approximates the probability density function of a set of input vectors. The output units form an ordered set of clusters of vector centers that sample the input space. The units compete in a modified winner-takes-all manner, such that the unit whose weight vector generates the largest dot-product with the input vector wins.

Competitive learning is also an unsupervised learning procedure. A set of hidden units compete with each other to become active, such that disjoint clusters of similar input vectors are formed [23]. As input vectors are presented, the hidden unit with the greatest total input wins and turns 'on' fully, while all other units turn 'off'. It then increases its weight vector by a small fraction so that in future trials it will continually win the competition for the same input vector. Constraints are placed on weight vectors such that the sum of its components are kept constant so that the same unit can't keep winning. Competitive learning can be modified to perform Kohonen

[2]There are times when a certain task requires continuous adaptation of the network, such as a system that changes with time.

feature maps simply by updating the weights of neighboring clusters.

Another unsupervised learning procedure based on competitive learning is called the Adaptive Resonance Theory (ART). This theory, developed by Grossberg and Carpenter [25], is a complex set of training rules for a feedback network; training a feedback network is difficult because adjusting weights affects signals in both the forward and backward paths. ART networks self-organize to stable recognition codes in response to inputs of arbitrary sequences. Suffice it to say that ART uses a complex modification of the competitive learning model; the technical details are beyond the scope of this section.

Modifications to the Hebbian learning rule are resulting in new learning paradigms that are becoming quite popular. Hebbian learning can be described as 'learning by experience' whereby neighboring units strengthen or weaken connections between themselves. The basic equation in this type of learning is $\Delta w_{ij} = \eta o_i o_j$, where o_i and o_j are the outputs of the i^{th} and j^{th} units, respectively, Δw_{ij} is the connection strength between these two units, and η is the learning rate.

A Linsker network [26] is a self-organizing, unsupervised learning paradigm that uses modified Hebbian learning in a linear layered network. This type of learning maximizes the variance at **each** unit's output such that maximum information is preserved at a layer of units, subject to certain constraints.

By combining both Kohonen's and Grossberg's learning rules, Hecht-Nielsen has come up with what he calls *Counterpropagation* [27]. It is not a learning rule in the strict sense, but rather it is a self-organizing network of combined rules. The network functions as a statistically optimal self-programming lookup table.

Reinforcement learning procedures are not as common as (un)supervised learning procedures; we are, therefore, only considering one type of reinforcement procedure here. The Associative Reward-Penalty (A_{R-P}) is an attempt to map automaton theory onto connectionist networks. Each stochastic unit (see Boltzmann below) is treated as an automaton and the state each unit adopts is treated as the automaton's action. If linearly independent input vectors are presented, and the network contains only one unit, A_{R-P} finds the optimal weight values. A network of these units will develop useful state representations [23].

Supervised learning requires that a *teacher* present a set of training examples such that the network can *learn* to recall these examples and make inferences about incomplete or unanticipated examples. A Boltzmann learning procedure, which uses a generalized Hopfield network, utilizes a *stochastic* decision rule to update the states of units. A probability distribution is 'learned' by adjusting the weights between units. Each unit has an associated probability that it will be in one of two states. Repeated applications of this rule to each unit will cause the network to reach 'thermal equilibrium'. At this point, the probability of a global state is constant and follows a Boltzmann distribution. The network first receives an input/output training pair and simulated annealing (a gradual change from one state to another to reach a global minimum) is used to reach equilibrium. Weights are *incremented* by Hebbian learning at this stage. This is repeated for the entire training set. A second phase is run with no output values and is again annealed to equilibrium. This time, however, weights are *decremented* by Hebbian learning (a form of *unlearning*). Learning halts when the probability distributions of the two phases are the same. A special case of Boltzmann learning, used in an unsupervised capacity, is when no *input* units are used. The network learns to model a probability distribution that is specified in the output units. The advantage with this type of network is that it will perform completion of a partial vector that is clamped at the output. Learning in a Boltzmann machine is very slow since reaching thermal equilibrium by simulated annealing is a time-consuming process. An approximation to the Boltzmann statistics, by *estimating* the network's *mean* behavior, is a solution. Mean Field Theory (from Physics) replaces each unit's binary state by a deterministic real-value that represents its expected value. Deterministic relaxation is used in place of simulated annealing, to obtain a representation of the equilibrium distribution [23].

We have been necessarily brief in our discussion of learning procedures thus far. Much of the mathematics have been omitted in favor of presenting conceptual ideas. We will now discuss, in more detail, the Generalized Delta Rule or Backpropagation Learning Rule.[3] Refer to **Appendix A** for a complete derivation of this Backpropagation Learning Algorithm. This supervised learning paradigm is, by far, the most common in its class. Backpropagation is the *workhorse* of all neural network learning paradigms. Backpropagation seems to be the best general-purpose model for generalization in a feedforward network that uses the Generalized Delta Rule [21].

The history of backpropagation is quite amazing. Although McClelland and Rumelhart et al. [2, 3, 85] have been credited with the inception of the backpropagation learning rule in 1986, they cited Parker's work on learning logic [28] from 1985. Later, Parker discovered the 1974 work of Werbos on regression analysis [29]. It turns out that backpropagation is a special case of traditional statistical regression

[3]The Generalized Delta Rule is a generalization of the Delta Rule which happens to be similar to the Least-Mean Square (LMS) Learning Rule [18]. Backpropagation seems to be a term coined by the PDP Group [2, 3]. Nevertheless, the two terms are used interchangeably.

techniques [30]. In 1988, LeCun [31] discovered the 1969 work of Bryson and Ho on applied optimal control [32], where similar techniques to backpropagation were apparently introduced. Who knows, "even earlier incarnations may yet emerge." [33]. Nevertheless, McClelland and Rumelhart can be credited with introducing backpropagation into a *usable* form for neural network learning.

Backpropagation learning requires the propagation of error deltas (differences between target and actual output values) backwards through the network. The application of the backpropagation learning rule to neural networks thus forms an apparent paradox. Real neurons cannot run backwards (evidence suggests this), so this learning rule is not biologically plausible. Also, **back**propagation learning is generally used by feed**forward** neural networks—the ultimate contradiction. However, it is only the *error* signals that are propagated backwards through the network during learning; and those signals propagate down the *same* connections (synapses) as the forward mode.

The three-layer feedforward network of **Figure 2.6** shows an overview of the backpropagation learning rule. The rule is, however, generalized to operate on any multi-layer network. We will assume the convention that o_j refers to the output of the j^{th} neuron and w_{ij} refers to the connection strength from the i^{th} to the j^{th} neuron [a superscript in the equation indicates the associated layer: I (input), H (hidden) and O (output)]. A set of training patterns is presented to the network one at a time. The input vector propagates through the network to form the normal sum of products, $net_j = \sum_i w_{ij} o_i$, where net_j represents the total input to neuron j. The output of neuron j is a function of its total input, $o_j = f(net_j)$. This function must be non-linear with a bounded derivative. The most useful transfer function is the sigmoid,

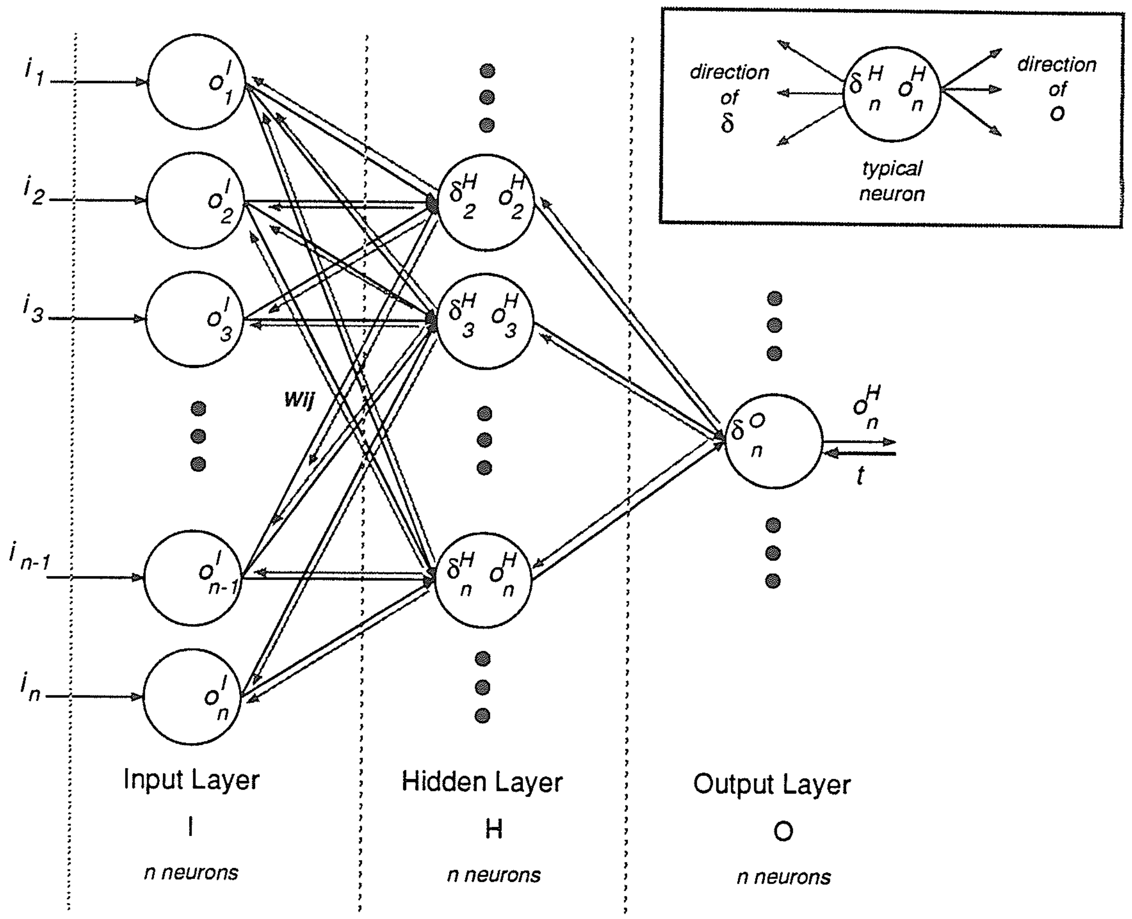


Figure 2.6: **Overview of Backpropagation Learning Applied to a 3-layer Network**

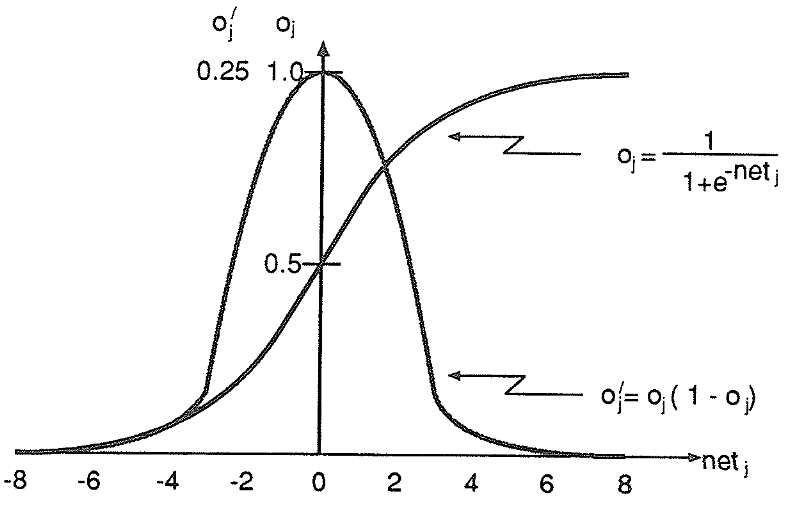


Figure 2.7: **Sigmoid Function (and derivative)**

$o_j = \frac{1}{(1+e^{-net_j})}$, shown in **Figure 2.7**, along with its derivative. The derivative of the sigmoid function is simply $f'(net_j) = o_j(1 - o_j)$.

The goal of the backpropagation learning algorithm is to reduce the total error, $E = \frac{1}{2}\sum_p \sum_j (t_{pj} - o_{pj})^2$, by adjusting the weights w_{ij}; where t_{pj} represents the target value of the j^{th} neuron with respect to training pattern p. Usually E is estimated by taking the error after each pattern and is averaged over the entire set of patterns. Presentation of an entire set of training patterns is called an *epoch*. Error derivatives are propagated back through the network from the output layer. An error vector is created at the output layer by taking the difference between the actual and target output vectors, $\frac{\partial E}{\partial o_j} = (t_j - o_j)$, where t_j represents the target value for the j^{th} neuron. Thus, the error delta of an output neuron is specified as $\delta_j^O = \frac{\partial E}{\partial o_j} f'(net_j) = (t_j - o_j)o_j(1 - o_j)$. The error delta traverses the network in reverse through the synapses to lower layers. We recursively define the error delta of a hidden layer neuron to be $\delta_i^H = f'(net_i)\sum_j \delta_j^O w_{ij}$. This definition can be used for any number of hidden layers in

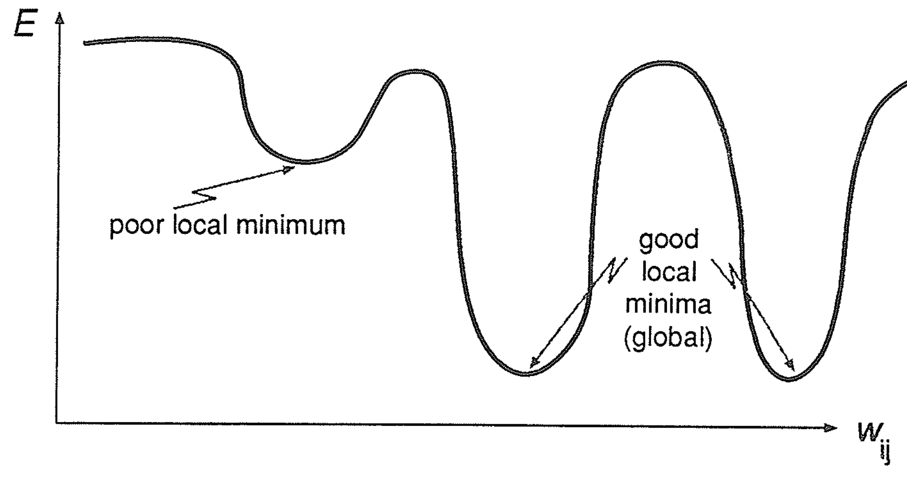


Figure 2.8: **Error Surface with Local Minima**

a multi-layer network. Learning is accomplished by modifying the weights according to $\Delta w_{ij} = \eta \delta_j o_i$, in the direction to decrease E. The process is repeated for the next training pattern and so on. After a number of epochs, a performance measure is taken and training halts when the error is no longer significant; i.e. when the patterns can be recalled adequately. The backpropagation rule is performing a gradient-descent heuristic as it tries to minimize the error function. As long as the training vectors are appropriate for a solution to exist, backpropagation will always find a set of weights that will minimize the error function, E. If the network has hidden layers, however, this might not always be a *global* minimum. Gradient-descent follows the negative slope of the error derivative. The error surface may contain many local minima; a 2-dimensional cross-section of an error surface is shown in **Figure 2.8**. If the set of weights finds one of the local minima, it does not mean that the solution is wrong — there could be many solutions to a single problem and the algorithm has just found one of them. Backpropagation learning rarely encounters poor local minima, provided there are more neurons and synapses than the problem requires.

Generally, the most serious problem with the backpropagation learning algorithm is that it takes many epochs, sometimes thousands, to converge. But backpropagation is great at training a neural network to generalize, therefore, much research has been directed into backpropagation *speedup* techniques. Since the conventional means of executing the backpropagation paradigm has been on sequential computers, the obvious way to speed up learning is to distribute the processing of each neuron (or even each connection) on a separate processing element. It has been shown [23] that the learning time for the backpropagation algorithm is approximately $O(N^3)$ where N is the number of weights in the network. A parallel distributed processing structure of weights would decrease the learning to approximately $O(N^2)$. The improvements will be significant for networks with a small amount of neurons, but not for larger networks. Alternately, we could implement a dedicated array of *hardware* neurons and synapses that would execute the learning algorithm directly.

By including a term as suggested in [2, 3], such that $\Delta w_{ij}(\mathrm{n}+1) = \eta\delta_j o_i + \alpha\Delta w_{ij}(\mathrm{n})$ (where n indexes the presentation number and α is a constant), 'momentum' is built up as the weight changes step along short regions of constant gradient in the error surface. This acceleration is cancelled-out as the error derivative increases.

Gradient-descent methods (first derivatives) only give the derivative of the error surface at a point. Knowing the curvature of the surface at that point gives us a better sense of direction. The second derivative $(\frac{\partial^2 E}{\partial w^2})$ would give us an estimate of the curvature and with this we may estimate the cost of removing some connections [34].

Some training vectors can pull the weights in opposite directions at each cycle thereby cancelling weight updates. In this case, we may speed convergence by constraining weight changes such that new vectors only minimally degrade the response of a previous vector [35]. Alternately, we may preprocess the training data to remove any correlation in the input vectors, i.e. orthogonalize the inputs.

Backpropagation learning specifies that a weight update is proportional to the forward propagated output and the backpropagated error, i.e. $\Delta w_{ij} = \eta \delta_j o_i$. But in hidden layers, the modification of weights to lower layers will cause an error in the *actual* neuron output, o_i. If we use the *expected* value of the output, we get a more accurate weight update given by $\Delta w_{ij} = \eta \delta_j (o_i + \delta_i)$ [36].

As you can see, there is no shortage of ways to improve the backpropagation learning paradigm; however, each of the ideas suggested here are limited with respect to the class of problems that they improve performance on.

2.3 Artificial Neural Networks

There are currently many neural network hardware designs that use traditional electronic circuit components and the more technologically advanced use of VLSI design. Most of these designs simply implement a specific neural network architecture with no real power to execute the learning paradigm that they were destined for. In these circuits, the designer has opted for an *easier* approach to neural network systems, and has allowed a computer to do off-line processing of the data.

Designing a circuit for VLSI implementation requires many decisions that are not necessary in discrete component circuit design. The critical issues are area constraints, speed, and processing technology. Some minor issues are CAD tools and standard cell libraries.

The use of analog processing components in VLSI, in respect to neural networks, has many advantages and disadvantages. A fully analog neural network can closely model the biological processes that are taking place [37] and can exhibit tremendous speed and capabilities in area efficient designs. However, analog VLSI suffers from some major problems. Analog integrated circuits suffer from what can be termed 'processing imperfections', which are due to manufacturing variations. These imperfections can manifest themselves into poor quality integrated circuits with unmatched characteristics not only from different processing runs on completely different pieces of silicon, but all the way down to separate circuits on the same wafer or even individual devices in the same circuit. It is true that large companies with huge budgets can afford to pay for higher accuracy in processing technology and can command special fabrication techniques, but the average researcher does not have access to, or the guarantee of, these kinds of facilities.

Most analog neural networks utilize storage capacitors (analog memories) to implement synaptic weight connections. Recent designs, like [38], that rely on capacitors for weight storage also use the leakage current to emulate the synaptic weight change in training and therefore require highly accurate devices. Similar analog networks, such as [39] and [40], also use capacitors for synaptic weights and require constant refreshing of voltage levels unless the chips are cooled to ridiculous temperatures like that of liquid nitrogen ($77K$) — even then the storage would only last a few days. Using capacitors is only one possibility for weight storage in analog neural networks. The use of custom programmable resistor chips [41] for the analog synaptic weights can also suffer from variations in device physics. Many other designs are using EEPROM technology [42] with much success; in fact, INTEL Corp. has just released an analog IC that uses EEPROMs for synapses. In all analog networks the devices must operate in the subthreshold region of the device transfer characteristics and, therefore, with limited supply voltage ranges in certain technologies, the available levels for weight values can diminish to the equivalent of 4-bit digital resolution (i.e. 16 levels) [43]. There are many other analog implementations of neural network paradigms along with hybrid designs that use the accuracy of digital storage with the speed and efficiency of analog arithmetic [44] as well as pulse-stream arithmetic processing [45, 46]; but these usually require fast, accurate, and area-consuming digital-to-analog and analog-to-digital converters.

On the other hand, digital electronics in VLSI is a mature technology that offers high accuracy (more precision), better reliability in functionality, noise immunity, and is easily manufactured. There are many realizations of digital weight storage, including RAM, ROM, EPROM and EEPROM, and shift register/latches. However, digital designs require significant silicon area to implement arithmetic processes, especially in fully parallel designs. Also, the speed with which these processes can be executed can be many orders of magnitude greater than similar analog implementations. Since almost every neural model contains neurons that compute weighted sums, it is advantageous to select an architecture that will reduce the area-time metric to a reasonable level. Ultimately, the architecture chosen will reflect a suitable solution to this metric. There are no shortages of possible digital architectures for neural network paradigm implementations. We will now concentrate mainly on digital VLSI neural network research architectures as reported in the literature. Some interesting digital *systems* will be mentioned for completeness; however, these architectures are well beyond the scope of this thesis. We also examine some current commercial digital neural network integrated circuit chips because they contain some interesting architectural ideas.

Digital systems for Artificial Neural Networks (ANNs)[4] can be broken into three broad categories [20, 47]: (1) General-Purpose Parallel Computers for Neural Network Simulations; (2) Special Purpose Processors for Neural Network Simulations (Neurocomputers); and, (3) Dedicated Digital ANN VLSI Circuits. All three categories present some interesting and valid ideas; however, we have set out to design a digital VLSI ASIC, and comparison to current VLSI research is a must. The general-purpose and special-purpose structures can be hierarchically analyzed to gain insight as to their inherent structure, and possibly present us with some ideas for our own design.

General-purpose parallel computers consist of programmable processor arrays for simulating a wide range of neural network paradigms in a framework that is analogous to conventional computers. The parallel processing computers optimize the neural processing by distributing the problem to a large number of simple processors. These types of architectures, although very flexible, can be very difficult to efficiently program and partition a neural network paradigm onto the parallel processors. The consummate pioneer of general-purpose parallel computers is the Connection Machine (CM) [48]. With its simple processing elements (PEs) and local memory, thousands of independent PEs operate concurrently on different segments of the same problem. The architectural concepts for the CM originally came from the connectionist theories, but the CM was destined as a general purpose *alternative* computing

[4]We tend to use the phrases "artificial neural network" and "neural network" interchangeably throughout this thesis even though an *artificial network* implies a man-made electronic architecture and *neural network* implies the biological representation of same. Suffice it to say that the context these terms are used in will make the representation clear.

structure to traditional serial Von Neumann-style architectures. The CM requires a specially designed programming language to distribute the processing power. A relatively similar architecture to the CM, the AAP-2, described in [49], is a massively parallel cellular array processor based on VLSI technology. It contains 65,536 one-bit processors that, like the CM, offer high-speed data processing in a Single Instruction Multiple-Data path (SIMD) architecture. A somewhat different approach to massive parallelism in a general parallel processor can be found in the Warp Machine [50, 51]. Warp's architecture incorporates a systolic array of 10 processing cells. Each cell consists of an adder, multiplier, and ALU, and communicates with its immediate neighbors only. A programming language was also developed to partition the problems that the Warp Machine was to simulate. The NCUBE microprocessor-based supercomputer [52] is also a massively parallel computer with processors configured in a 10-dimensional hypercube;[5] the CM uses a 16-dimensional hypercube. Both the NCUBE and the CM contain the same architecture; however, the NCUBE processes data in a Multiple-Instruction Multiple-Data (MIMD) fashion. There are many other general-purpose parallel computing architectures [20]; we have only mentioned a few for the sake of completeness and because of their inherent similarity to basic neural network structures. These machines are all characterized by a large number of simple processors that are connected in various topologies in a massively parallel fashion.

Architectures such as these are well beyond the scope of this thesis but are worth discussing because the basic concepts used in performing massive parallelism are relevant to any neural network structure. However, these structures suffer from the

[5]An n-dimensional hypercube connects a single processor to n of its neighbors and has the advantage that the links of the hypercube provide almost the same communication capabilities as a fully connected network that forms a complete graph.

difficulty required in scheduling processors for the high interconnection needs of neural networks. In our opinion, these hardware specific programming languages are an essential and critical part of general purpose parallel computers and unless a design can develop an optimum data partitioning scheme for multiple paradigms, these computers will not be a major neural network hardware influence.

In contrast to general-purpose parallel computers, we present the special-purpose processors designed especially for neural network simulations. These *neurocomputers* usually consist of special hardware boards optimized to perform neural network paradigms that interface to traditional sequential computers. The hardware involved can range from simple modular processing elements to Digital Signal Processors (DSPs) integrated into serial or parallel structures either dedicated to executing specific paradigms or, more commonly, generally programmable to execute multiple paradigms. There is a vast array of both commercial and research projects that can offer tremendous speed increases for neural network simulation; but these boards are mainly DSP boards optimized to perform the weighted summations inherent in most neural network paradigms. Most of these coprocessor boards are designed to interface to IBM-PCs or Sun workstations through the VMEbus. Also, most of the boards discussed here use industry standard microprocessors such as the MC68020 (Motorola), or DSPs like TMS32020 (Texas Instruments). The virtual processing elements and interconnections established during run-time to implement a paradigm require huge amounts of memory, and the usual multiplexing of processors takes away the processing power inherent in the parallel structure of neural networks. The systems that offer some novel structures or that *stray from the norm* will be presented in more detail; however, we will mention many systems for the readers who are interested in simulations of neural network paradigms because some of these are readily available, easily attainable, most adaptable, and highly affordable.

Some commercially available neurocomputers are systems by Hecht-Nielsen Neurocomputers (HNC), Science Applications International Corporation (SAIC), and TRW—to name a few. The Anza Plus/DP from HNC [53] is a neurocomputing co-processor board available for the IBM-PC or Sun. The architecture is based on a 4-stage pipelined Harvard architecture employing separate data and instruction paths. ANZA uses a specialized floating point DSP to achieve speeds of 3 Million interconnections Per Second (MiPS)[6] during training and approaching 12.5 MiPS peak performance with 64 bit accuracy. Software is also available to be used in conjunction with the board to provide a user interface to the extensive set of neural network paradigms that HNC provides. SAIC has developed a set of neural network processing boards, Delta Floating Point Processor, described in [54, 55], that use fast static column mode memory and an ECL (Emitter Coupled Logic) floating-point processor chip (from BIT) that implements a Reduced Instruction Set Computer (RISC) architecture. The Delta also uses a multi-stage pipelined Harvard architecture that claims simulation speeds upwards of 3 MiPS during learning and 11 MiPS without learning. A third commercial neural processor board has been designed by TRW and is called the Mark IV neurocomputer [47]. TRW designers used a Motorola 68020 microprocessor and 68881 floating-point co-processor configured such that virtual PEs are 'swapped in' in the neural processing phase—much the same as the concept of physical memory in virtual memory computers. Large neural networks with many interconnections and speeds approaching 5 MiPS are obtained by this architecture.

[6]Not to be confused with Million Instructions Per Second (MIPS).

It is interesting to note that these architectures all use a single, fast special processor with no true parallel processing power.

We now turn our attention to systems developed for research purposes. One of the most interesting and daring projects is Netsim by Garth [57]. The system was designed to accelerate learning in neural network paradigms by using a small instruction set, pipelined operations and large amounts of memory. The computational speed is gained by the use of two custom-designed, specialized co-processors. A communication coprocessor handles all interfacing of Netsim 'cards' into a *nearest neighbor* set, while the 'solution engine' (basically a vector coprocessor or an optimized DSP) performs mathematical computations optimized to neural network requirements. The network of cards which form a neurocomputer require a host computer acting as a system controller. This system exhibits speeds of nearly 90 MiPS during learning.

Another parallel neurocomputer is described in [58]. A parallel array of processing elements simulate a column of neurons in a multi-layer neural network. Each processor contains a TMS32030 DSP, memory and control. The processors are configured in a 1-dimensional systolic ring architecture that executes vector-matrix multiplication in a SIMD fashion. Again, this system connects to a host computer, such as Sun, via VMEbus, and obtains speeds in excess of 500 MiPS.

A neurocomputer along the lines of Netsim has been developed by Pacheco et al. [59], in which a custom coprocessing element has been designed to ultimately form a parallel MIMD computing structure. The coprocessor is designed under the RISC architecture and is basically a microprocessor (ALU, registers and memory).

In a more abstract design philosophy, the principle of communicating concurrent processors has been developed into a neurocomputing architecture of multiprocessors by Kraft and Frostrom [60]. The idea here is that processing resources are shared between processing elements, through something called *actors*, with distributed communication. The system acts analogous to a post office in that interprocessor communication guarantees nothing other than *mail delivery*; timing and order of processing is not guaranteed.

The systems developed thus far, are all attempts to increase neural network simulation processing speed by implementing current technology as a traditional *co-processor* or by custom designs of optimized DSPs configured as neural network systems. We have presented only a few such systems, but more can be found in the literature [20, 47].

A wide and diverse range of digital VLSI neural network ASICs have been developed — from simple logic gates to wafer-scale integration. The main goal in all implementations is to form a massively parallel set of basic electronic neurons and synapses (the biological representations) in a highly efficient structure and *place* as many of these elements on the silicon as is physically possible. There are no shortages of design strategies in digital neural processing as designers strive to develop the 'microprocessor of neurocomputing'. Although most of these designs have no facility for on-chip learning (all learning is usually done off-line by a host computer), these designs show innovative architectures using state-of-the-art technology in an approach that can rival the *speed* and cost of analog devices. In reviewing these architectures, we hope to gain knowledge that we can build on and apply to our own design.

The architecture of a neurocomputer microprocessor is described in [61]. The chip consists of a number of simple processor nodes (similar to DSPs) connected in an SIMD configuration. Each node is a simple arithmetic processor that performs vector-matrix multiplications. Multiple nodes form connection layers that emulate a neural structure. The communication between nodes is via "broadcast" structures to cut down on interconnection area. The processor nodes are 16-bit two's complement multipliers with associated registers, logic-shifters, memory for weights, and an adder.

A general purpose digital architecture for neural networks has been developed by Duranton and Mauduit [62]. The "neuromimetic chip" is a fully parallel design wherein multiplications are serialized by replacing multipliers by AND gates and accumulating the results to form a weighted summation. The designers intended for this chip to be used in a Transputer array for general purpose network simulations.

Hirai et al. [63] designed a digital neuro-chip for large scale neural networks. The chip contains six neurons and 84 synapses with 64 levels of modifiable weight. The designers opted to use impulse density encoding of digital data to obtain signals with the *flavor* of analog (continuous) values.

This encoding scheme is similar to the "pulse stream" arithmetic described by Murray et al. [5, 6, 45]. In this scheme, neural activities are represented by a stream of digital pulses (similar to natural neurons) on excitatory and inhibitory data lines. The data is decoded by using "chopping clocks" that basically integrate the signal to form smooth activation potentials similar to a "ring oscillator".

Murray et al. also describe another architecture for neural networks involving bit-serial arithmetic. The bit-serial design involves simple processors communicating with each other via single-bit data lines. This structure exhibits computational efficiency as well as tightly pipelined arithmetic processing. A simplified 5-state activation function allows shifting and adding in place of the silicon-*hogging* multipliers. Bit-serial addition can be utilized to create highly efficient pipelined multipliers. This type of arithmetic processing is desirable in neural network structures of massively parallel processors, since the data is communicated between processors on a single-bit data bus. We will expand more on bit-serial architectures in the development of our own neural network design.

Along the lines of impulse density, there are many VLSI neural network designs that employ stochastic processing. This type of processing involves the use of statistical data structures using probabilities. Basically, the circuits represent digital values as analog probabilities of pulse densities — the number of pulses in a particular time interval. These designs differ in the way in which the probabilities of pulses are summed together to form the mathematical computations of neural processing. All stochastic processing requires the use of random numbers to generate these probabilities and would benefit greatly from the use of pseudo-random number generators such as Hybrid Cellular Automata (HCA); although none use HCAs (all pseudo-random number generation in stochastic processing is currently done by computer). Van den Bout and Miller, along with other colleagues, have developed many stochastic implementations [64, 65, 66, 67, 68], which all use the same basic technology. Synaptic weight shift registers cycle through their values which are compared to a random number to create a stochastic pulse stream with a probability of being 'on' proportional to the digital weight stored. Multiplication is simply performed by AND-ing this stochastic pulse stream with a stream representing the neuron activation, and summation is done by digital counters. Off-line computers are utilized for neural transfer function processing. Various stages of pipelining are introduced to create bit-level processing.

A very unique approach to stochastic processing has been implemented by Tomlinson et al. of Neural Semiconductor, Inc., and a proprietary architecture is described in [69]. Tomlinson has analyzed the stochastic structure and found that a simple OR-ing of excitatory and inhibitory pulse trains can simultaneously provide the weighted summation **and** the non-linear activation function required by some neural network paradigms. The system offers a very high density of digital neurons and synapses with an analog-type performance. However, probabilities that are statistically independent are required for all types of stochastic processing; therefore, good random number generators are required, with long iteration cycles and these systems **can** be very slow. Another proprietary design of a digital neural network is the neural bit-slice computing element developed by Micro Devices [71]. This bit-slice design utilizes serial adders and multipliers in a highly multiplexed configuration.

A set of digital neuron-type circuit elements has been developed by Habib and Akel [70] that are based on simple logic gates. A neuron-type processor is constructed by cascading a 'cell body', an 'axon base' and an 'axon circuit'. Each of these elements are constructed from simple AND or OR logic gates. Processing is done by spatial and temporal processing of pulsed data values.

Another simple VLSI structure is a connectionist architecture designed by Cleary [72]. This design uses single-bit data and heavy multiplexing to obtain 1000 input lines (synapses) distributed over 100 output lines (neurons). A host computer cycles the massive array of single-bit weight shift registers (which take up the bulk of the chip) and processing is accomplished by AND-ing inputs with a mask of weights and counting the resulting number of ones.

A multi-layer perceptron using discrete weights is described in [73]. Multiplication is performed via shifting and adding and weight values are confined to powers-of-two. Learning is performed off-line and activations use a lookup table. Binary logic is also used in a programmable logic approach to neural networks [74]. Dynamically Programmable Logic Modules (DPLM), as opposed to field Programmable Logic Arrays (PLA), allow the dynamic reconfiguration of functions in a PLA structure. A "tree of DPLMs" are used to implement a multi-layer perceptron.

Systolic architectures are very promising in implementing large networks because of their high regularity, local communications, and inherently parallel processing structures. A ring systolic design for artificial neural networks [75] proposes a systolic communicating array of processors that can efficiently execute many neural network paradigms. A second systolic architecture [76] implements the backpropagation algorithm in an array of regular, nearest-neighbor processors. A complicated systolic cell of a digital multiplier, adders, and many multiplexors form the basic element that completes a single layer array of a multi-layer backpropagation network on an ASIC. External multiplexing and multiplication is necessary to complete a multi-layer network.

In a design a little closer to home, Diamond et al. [77] have developed a neural network architecture imbedded in a fuzzy cognitive system. The neural network contains a simplified multiplication scheme using logic gates and utilizes external weighting, along with recursive pipelining of addition, to perform a weighted summation. The overall accuracy seems to be 4-bits; however, internal processing is done at 10-bits resolution. Huge shift registers cycle their data to accomplish the multiplexing of inputs to create a neural network layer.

Finally, on a larger scale, Wafer-Scale Integration (WSI) is presented. A WSI system optimized for neural networks using completely digital circuits is described in [78, 79]. The architecture employs a time-shared bus which implies that only one 'synapse' is needed per 'neuron'. The 'neuron' is a standard microprocessor design in a custom ASIC while each 'synapse' is a simple interconnection—the synaptic weights are stored in a high-speed memory structure within the 'synapse'. A fully connected network is realized by a tree structure of 'neurons' combined into a neuron ASIC. Each of the neurons are interconnected on the silicon wafer to form a larger network.

There are many other digital VLSI ASIC designs for artificial neural networks [80]. We have presented a large sampling of what we consider to be the most interesting from both a VLSI design standpoint and a connectionist viewpoint. There are many ideas for ANN designs and it may take awhile for one to digest the ideas presented herein. However, one can see that the field of neural networks has progressed tremendously, and that there is a sea of ideas that a connectionist can dive into—and, if he/she doesn't keep his/her head up, he/she could easily drown.

Chapter 3

Implementation and Results

We have seen how artificial neural networks have been used in real-world applications and the power that they possess to do many problems that are viewed as **difficult** by traditional means. We have also seen how other researchers have implemented neural networks in both hardware and software. Similarly, we have shown examples of hardware implementations from a system level to silicon. In this chapter we will concentrate on the development of a neural network in a VLSI ASIC that will perform the backpropagation learning paradigm. The inclusion of on-chip learning capabilities enhances the novelty of the device from a neural network perspective.

The development of an ASIC provides the designer with total freedom in regard to a multitude of implementation strategies, and can be considered more of an art than a science. We don't mean to suggest that VLSI does not require any scientific skill, but we do want to stress the idea that an ASIC design requires somewhat of an artistic approach—something like a painter who can take a blank canvass and produce a beautiful landscape.

The time required to produce a design is proportional to the complexity of the problem and can grow to unwieldy limits; this is also true for hardware (space) requirements in VLSI. After an algorithm has been decided on, we must then consider all possibilities of implementing it. Design choices include available technology, Computer Aided Design (CAD) tools, simulation tools, digital or analog cells, and mathematical techniques. In this thesis we have decided to implement a neural network with backpropagation learning in a VLSI ASIC. The design uses bit-serial arithmetic techniques in a fully digital implementation with some unconventional VLSI layout techniques to obtain a highly efficient and functional network.

Area constraints in VLSI are a major dilemma that the complexity of a problem tends to overshadow. The backpropagation algorithm used in a neural network is fairly complex and thus a major VLSI design effort was undertaken to obtain a network with a relatively small number of neurons, that could execute a fairly complex set of events. We now present the VLSI ASIC design from its inception to its implementation. We showcase some architectural considerations along with a unique VLSI design strategy in the sections that follow. We conclude this chapter with a section on the actual design implementation.

3.1 Considerations

The previous chapter offered a sampling of current implementations of ANN systems. From a VLSI perspective, these examples have displayed a vast array of techniques and strategies in ANN design, and have given us a very diversified look at digital architectures. Of these architectures, stochastic, systolic, bit-serial and -parallel designs offer the most interesting and novel approaches.

Since the backpropagation learning paradigm requires the use of many multiply-and-accumulate functions, we will concentrate on the previously stated architectures only as they apply to the design of digital multipliers. This is a justifiable consideration, given the fact that the most area-consuming part (besides storage) of any digital VLSI design is the multiplier. We will thus concentrate on the design of multipliers and this will dictate the architecture for the remaining modules.

There are many multiplier design policies for digital signal processing [83]. Systolic multipliers act in a manner similar to the beating of a heart. A systolic implementation of the backpropagation algorithm was done by Kwan et al. [76]. The major disadvantage of this technique is that data is usually pipelined at different speeds and requires a multi-clock strategy. Systolic architectures, however, have a distinct advantage in that communication between processors is through nearest-neighbor interconnections.

Stochastic architectures [68] offer novel ideas by integrating probabilities over time. This has the advantage of analog-like computation but requires integration over long intervals of time. Good random number generators are also required for increased accuracy [69]. Pulse stream arithmetic is an alternative, however, elegant clocking schemes are necessary [45].

Parallel multipliers offer speed at the expense of area. However, the large area required by parallel bus structures and the clock speed degradation due to propagation delays make this alternative generally unappealing. Bit-serial architectures, on the other hand, offer high-throughput digital multiplication by pipelining operations. Since data is communicated between processors on a bit-wide bus, communication and routing overheads are minimized.

We must also consider the use of parallel and serial multiply *structures*. The use of a single parallel multiplier that is multiplexed throughout an ASIC, or the use of many serial multipliers working in parallel, are issues that must be addressed. A single multiplier can save considerable amounts of silicon area — this is an attractive possibility. However, we would like to keep the parallelism that is inherent in neural networks within our design. Utilizing many multipliers operating in parallel has the advantage of increased speed and will offer an architecture that more closely resembles an artificial neural network. These issues will be addressed further in the section on VLSI Implementation.

Parallel multiply structures can be considered the *norm* in current multiplier designs. Therefore, to test the ideas behind bit-serial architectures, we implemented a Hopfield-type neural network as described by Murray et al. [5, 6]. The bit-serial nature of our design was analyzed and the results were very encouraging. The multiply and accumulate structure of a bit-serial pipeline were thoroughly analyzed.

Implementation of the backpropagation algorithm in silicon requires high accuracy [40, 47]. For this reason alone, we have been concentrating on digital implementations as opposed to analog. Advantages of both technologies have been previously discussed. Further, we have decided that 8-bit accuracy will offer a good compromise between silicon area usage and dynamic range. The use of two's complement data representation is essential such that positive (excitatory) and negative (inhibitory) connection strengths may be realized.

Finally, the issue of on-chip learning is considered. On-chip learning requires the use of large amounts of memory and can consume valuable amounts of silicon area. The use of many storage devices is analyzed in the Implementation section. An on-chip activation function will add to the novelty of the design and should also be considered. A truly autonomous architecture will only be realized if learning occurs on-chip.

The next sections discuss the design and implementation. We have decided that the design of the individual modules will dictate our final implementation. Therefore, we will not state any specific design requirements other than what has been discussed above. We hope to obtain a network of neurons that will perform the backpropagation learning paradigm. How many neurons we can *squeeze* onto a chip will be discovered as the design progresses.

3.2 A VLSI Design Strategy

Just as there are many ways to execute a specific algorithm in silicon, there are equally as many techniques (coupled with specific technologies) that can be employed to implement physical devices in VLSI. The first thing to do after deciding that an ASIC is a necessity, is to take inventory of the available processing facilities and technologies, and CAD simulation and testing tools. The second step in any VLSI design is to partition the algorithm into functional modules that can be easily put together in a hierarchical structure. Of course, we are assuming here that the computational modules have already been finalized; that is, the way in which computations are to be executed (eg. parallel, systolic, serial, etc.). Usually, the ASIC designer tries to minimize the area of all structures in the design hierarchy so that the absolute minimum amount of silicon is used; silicon and processing costs are considerable, so this is highly desirable. However, there are often times when functional blocks of a design cannot be compacted or may not need to be; examples of these are: (1) if all blocks in a lower level are already compacted, any higher level compaction may be impossible; and, (2) if a design is relatively small, the extra effort involved in compaction time may not be justifiable. Therefore, there are two strategies to developing an ASIC design: (1) let the circuit design dictate the chip size (small designs); or, (2) let the chip size dictate the circuit design. By this we mean that a small design can usually fit on a minimum size silicon die, but larger designs have to be carefully planned at every stage in the design process so that they may fit on the maximum size die offered by the processing facilities. This is exactly what we must perform in our implementation of a neural network paradigm. We want to *pack* as many modules (i.e. neurons and synapses) onto a die that is physically possible, so we must constantly strive for compactness. The usual AT metric that is associated with VLSI design [84] must also be conformed to in order that the algorithm performs in an efficient manner in a reasonable amount of physical space. However, we encounter a second AT metric that equates physical design implementation time to the compactness of circuits; i.e. you can't spend all of your design time compacting layouts. It turns out, as you will see in the section on VLSI Implementation in Silicon, that our design implementation required large amounts of silicon area — therefore, special VLSI design strategies had to be employed. We feel that the design techniques discussed in this section are quite novel (although similar design styles have been used in CMOS3 Cell Library [10]) and could form the basis for a standard cell library of 'overlapping cells'. Of course, implementing a whole library of cells is beyond the scope of this thesis, so we present only the technique.

Our implementation of the backpropagation neural network paradigm originally began as one in which we were to use a schematic capture tool (such as Edge, by Cadence Design Systems) so that we could concentrate our efforts on the computational aspects of the algorithm, rather than on the physical VLSI layout of blocks within the entire structure. The University of Manitoba Electrical and Computer Engineering Department has a large library of standard cells that can be used and the CAD tools available would let us concentrate on obtaining an efficient design from a schematic level. A preliminary design of a Hopfield neural network using a similar design philosophy indicated good results. However, it became evident during the course of this design that we must either investigate different ways of implementing the mathematical calculations (eg. stochastic, multiplexed parallel processor, etc.) or come up with some unique device layout strategies. Since we felt that our design offered a novel approach to ANNs, we opted to investigate alternative design styles in the chosen technology.

Complementary Metal Oxide Semiconductor (CMOS) technology, which fuses NMOS and PMOS, was the design style of choice. CMOS offers the advantage of low power consumption with a disadvantage that approximately double the area is required as compared to similar NMOS and PMOS designs. However, since CMOS technology is offered by the Canadian Microelectronics Corporation (CMC) (where most Canadian universities get their designs implemented), and the University of Manitoba standard cell library contained an abundance of CMOS devices, we chose this technology. All of these standard cells are fairly well compacted and efforts on our part proved that there is no advantage in trying to further compact them. Therefore, we investigated the possibility of many alternative static and dynamic CMOS techniques such as pseudo-NMOS, clocked CMOS (C^2MOS) and CMOS domino logic [11]. Because a major part of our design area was going to be consumed by the storage of synaptic weights, we concentrated our efforts on the CMOS alternatives as they applied to storage devices. In many cases it was found that these CMOS alternatives didn't offer any area savings as they were applied to flip-flops[1]. Also, we felt it was not worth the extra effort that would be required to experiment with the ratioed device structures of these alternative schemes. These devices usually contain some type of precharge-discharge logic structure that requires elegant clocking strategies [12] or are plagued by a charge-redistribution problem that impairs their usability [13].

Again, since our design required major storage facilities (for example, a backpropagation network with N neurons would require N^2 synapses—with *m-bit* accuracy we would require mN^2 bits of storage for synaptic weights alone), we also investigated the use of Random Access Memory (RAM) in both static and dynamic modes. Dynamic RAM is very difficult to design and equally as difficult to process, so we only attempted Static RAM (SRAM). We produced layouts of SRAMs, from [11], and obtained only slightly better area advantages over a static CMOS D-type flip-flop (DFF) (with no set or reset) available in our standard cell library.[2] SRAM would require the use of extra control circuitry for precharge and address generation, as well as facilities for multi-port access (since our design is to operate in parallel and access to more than one value at a time is essential), and offered no real gain in area savings.

External storage in commercial RAM and investigation of a custom VLSI storage

[1]A flip-flop is a simple storage device.

[2]Note that commercial RAM manufacturers have special processing facilities to get high densities.

integrated circuit (IC) were considered, however, the communication overhead would have been tremendous. Also, the autonomous nature of the design would have been lost because on-chip learning would not have taken place. Therefore, we decided to concentrate on the use of static CMOS shift registers built from DFFs. The use of shift registers in our design also provided a 'natural' progression of data because of their inherent bit-serial architecture.

Because memory devices, such as RAM, exhibit a highly regular structure, the packing densities of these devices can be tremendous. Shift registers are also highly regular and it was felt that high density structures could also be achieved. A library of CMOS standard cells that can be 'stacked' side by side to form long dense arrays is described in [10]. The shift register cells are designed such that power (*vdd*), ground (*gnd*), and clocks run horizontally through the cell. The input and output are such that a large shift register can be realized simply by matching the output pin of one stage to the input pin of the next stage. Since our library of cells already had a similar layout style to these cells, we attempted to modify our shift register cells (DFFs) so that we could "stack" them to form the necessary data-holding shift registers in our design. We ultimately modified many cells in the library (for example, AND, XOR, and FADD), but we will only describe the design style for the modified DFF, as all strategies can easily be applied to other devices.

In order to obtain long shift registers, *N* DFFs can be cascaded to form an *N-bit* shift register. A simple static DFF is shown in **Figure 3.1**. Notice that the input to the device is on the diffusion layer; actually, it is the connection between n-type diffusion of the lower transistor and p-type diffusion of the upper transistor of the input transmission gate. Also notice that the output is on the diffusion layer of the

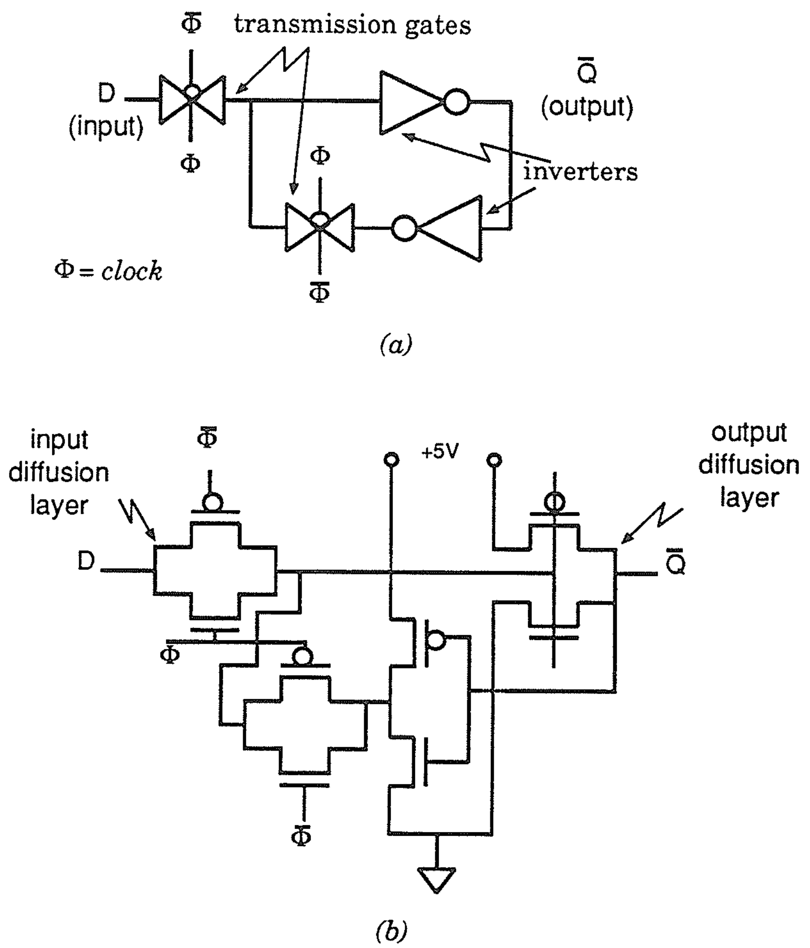


Figure 3.1: **Simple Static DFF:** (a) schematic level (b) transistor level

output inverter. Both n-type and p-type transistor pairs on these two devices are of similar ratios and, therefore, instead of using this cell as a standard cell,[3] we proposed the overlapping of input and output diffusion layers, as shown in **Figure 3.2**, when implementing a shift register. This allows maximum area utilization such that long shift registers can be formed. Careful placement of the input and output transistors are necessary such that the cells that overlap can share the diffusion layer. An area savings of 10% is achieved by this overlapping structure.

Using the above idea, we can create area efficient shift registers. Since an ASIC is limited in usable silicon area, we cannot make arbitrarily long registers. In our design we require a large amount of 8-bit shift registers that are shifting into each other in parallel; i.e. one 8-bit register feeds another and so on, and there are multiple devices as such. We therefore extended the overlapping idea into the sharing of power and ground rails, as they run horizontally across a cell, to even further increase the density of devices. Many problems occur when sharing power rails because a split-contact is usually placed on the p-well, near or at an n-transistor, to make sure that the well stays at ground potential, and another split-contact is used in the p-transistor area, such that the n-type substrate can be pulled-up to the power potential. This helps to prevent a detrimental effect, inherent in all CMOS processes, known as latchup.

Latchup is a temporary shorting of the well and substrate (usually clamped at *vdd* and *gnd*, respectively) due to a pair of parasitic bipolar transistors formed when the sources of n-type transistors are connected to ground, and p-type transistors to *vdd*. The split-contacts, shown in **Figure 3.3**, reduce the effects of the bipolar transistors

[3]A standard cell usually has a non-overlapping boundary such that devices are abutted side-by-side in a layout.

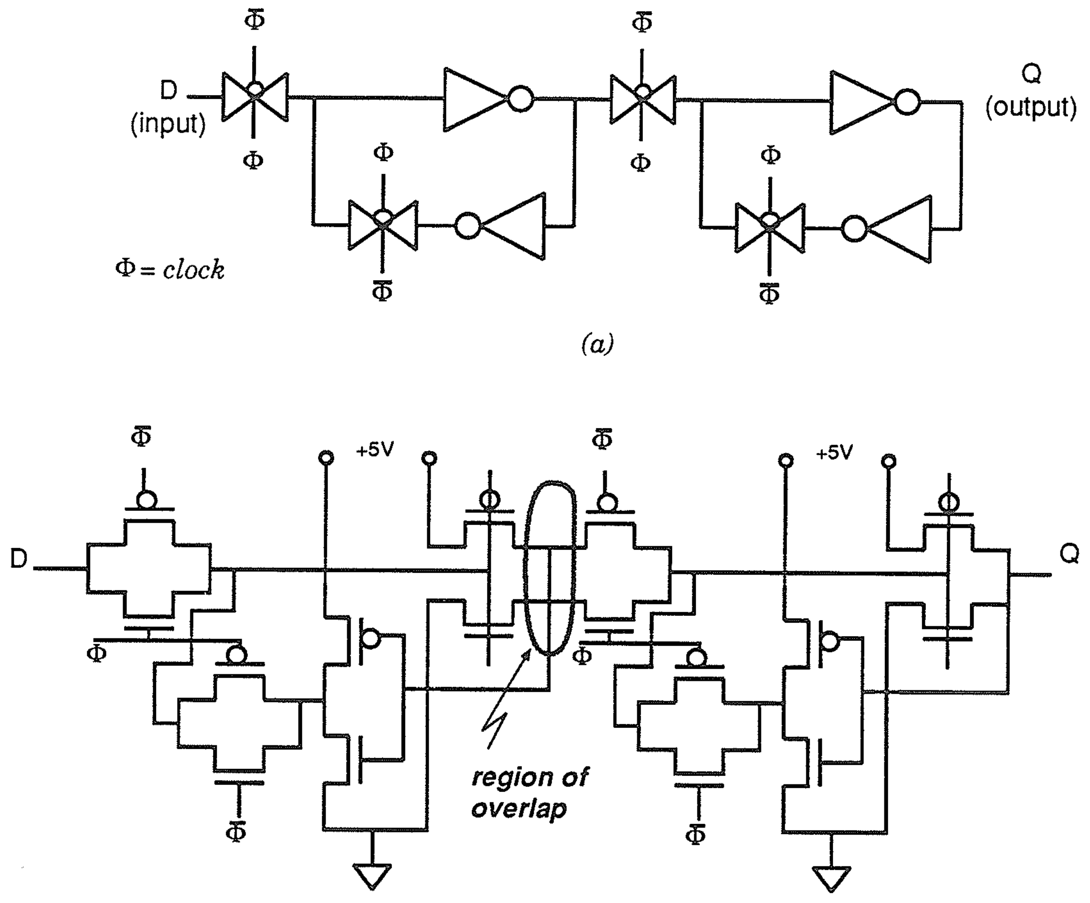


Figure 3.2: **Overlapping in a 2-bit Shift Register: (a) schematic level (b) transistor level**

and are always placed at regular intervals in a CMOS structure. If we try to overlap supply rails, we run into a dangerous situation when these contacts are overlapping; there is even more danger when trying to overlap cells of different logic as contacts can overlap in *unusual* places. Design rules must be followed such that when contacts overlap they don't become too big and ion-implantation regions don't come too close to each other. Careful planning of contacts can produce a *clean* layout with no design rule violations in cells of overlapping structures.[4]

A set of overlapping cells that are stacked vertically such that supply rails are shared is shown in **Figure 3.4**. An area savings of 15% is realized with such a structure, as compared with minimum separation standard cell placements. We have opted to route signals within (and through) cells to further increase the density by alleviating the need for routing channels. These stackable DFF cells are used in abundance throughout our design and the extra effort involved in laying out these cells has proven useful—as will be evidenced in the next section.

Using Edge, the Cadence CAD tool, for VLSI layout of these cells was accomplished via a simple array command that replicated the cell as needed. Edge was also used to hierarchically place lower levels of the design, and to modify the standard cells to make them stackable. The CAD tool is not intended for full-custom layouts, but with a little patience, a lot of practice and the use of manuals [14], a complete full custom design can be accomplished.

We previously stated that there is a danger in overlapping different types of cells; overlapping of different cells could create many unpredictable design rule violations.

[4]Note that a cell that has a design rule violation (but the overlapping of cells eliminates it) is legal as long as the cell isn't used alone.

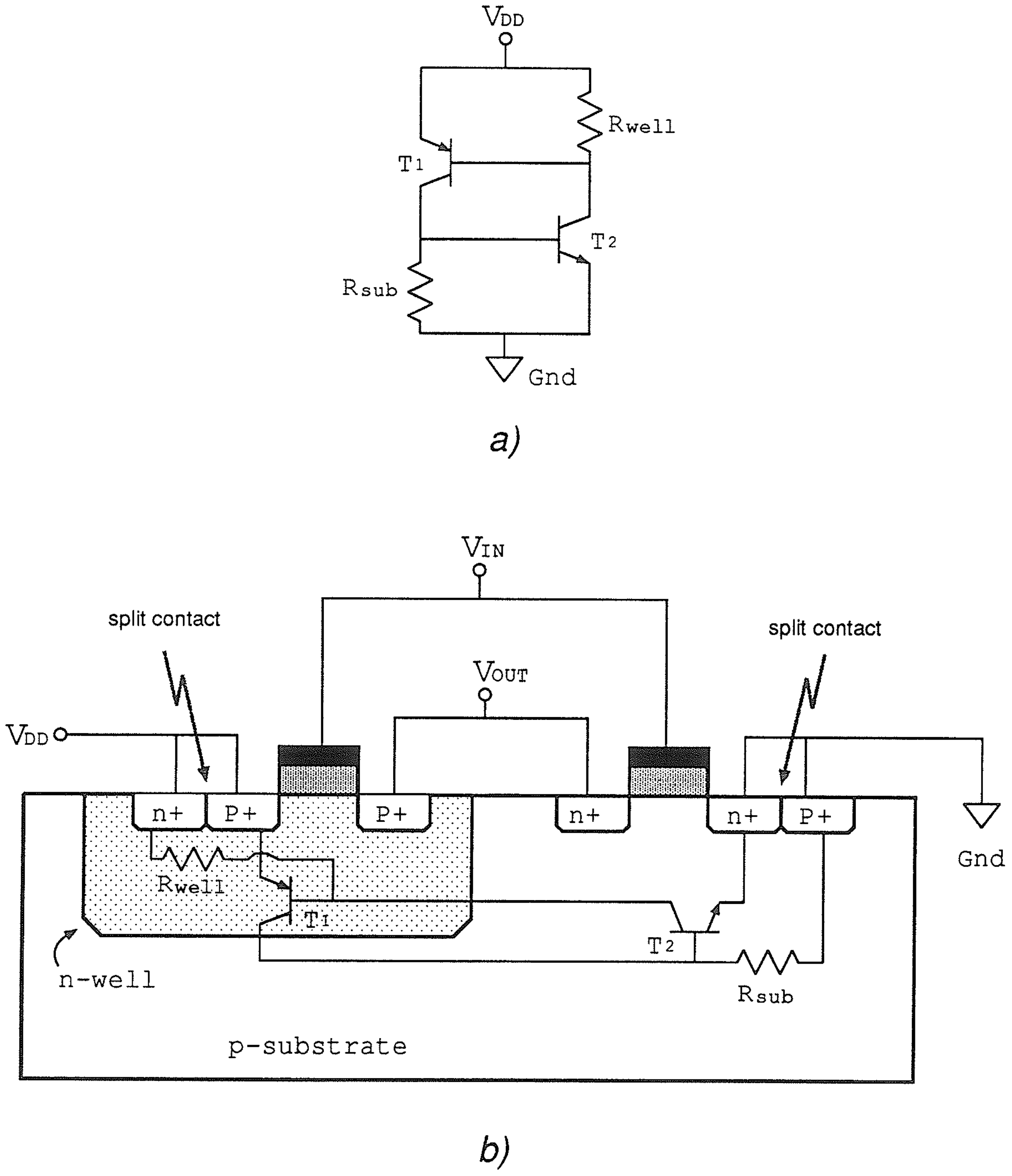


Figure 3.3: CMOS Latchup

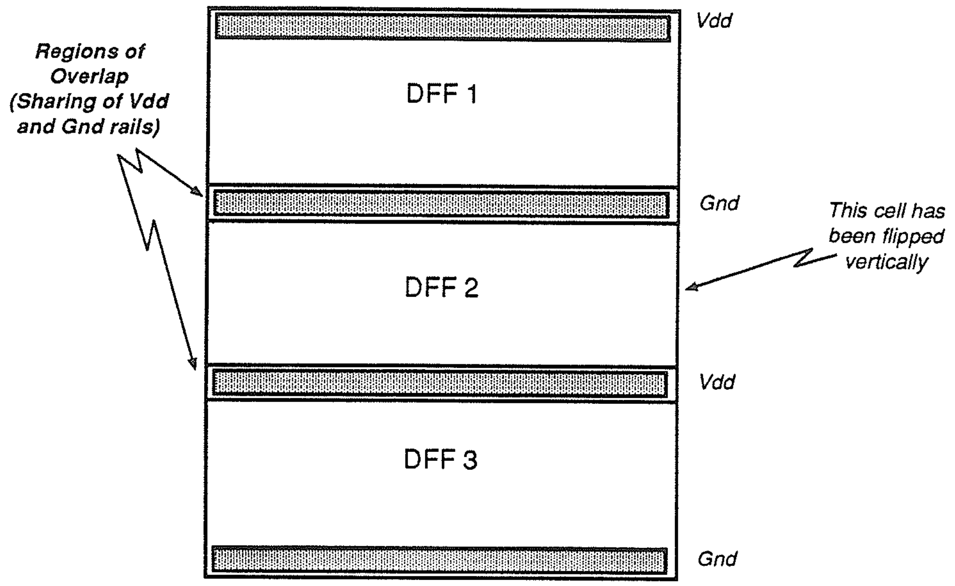

Regions of Overlap (Sharing of Vdd and Gnd rails)
Vdd
DFF 1
Gnd
This cell has been flipped vertically
DFF 2
Vdd
DFF 3
Gnd

Figure 3.4: **Stacked (overlapped) DFF Cells**

In response to 'good' results achieved for DFFs, a major design effort was undertaken so that many other cells could also be overlapped. Logic gates (such as AND, XOR and OR) as well as the full-adder (FADD) were completely modified so that they, too, could be used in this novel fashion. Another advantage of this type of structure is that all routing between cells can take place inside the cell boundaries so that no routing channels are required; routing is a major design constraint in VLSI. However, a disadvantage is that only highly regular structures are implemented easily; irregular structures, such as boolean logic functions, require greater attention. Fortunately, our design contains some regular structures and we may utilize this technique. It turns out that the backpropagation neural network paradigm, as we had implemented it, required the use of many *stacked* modules. In fact, the adder, multiplier, sigmoid generator, and weight register all used this technique and constituted a major portion of the design effort. The implementation of these cells is described in the next section.

3.3 Implementation in Silicon

The backpropagation neural network paradigm requires a network design that can actually run in *reverse*. That is, after the forward propagation of the input vector to the output layer, the error between the actual output and the target vector is propagated *backwards* through the network. The complexity of calculations is no greater in the backward mode as it is in the forward mode; there are just more equations to implement. The backpropagation equations can be found in **Section 2.2** and **Appendix A.**. An overview of the ASIC that will perform backpropagation is shown in **Figure 3.5**. There are N neurons and N^2 synapses which implement the basic concept of the algorithm by forming a single layer of a multi-layer feedforward neural network. This figure shows how input activations from lower layers are fed into the network through synaptic weights and how neurons create weighted summations of this vector and send an output activation to the next highest layer. Error signals are backpropagated in much the same way as forward signals, as is indicated in the figure. Each neuron (cell) functions as an independent module that can operate in parallel with all other neurons. This simplifies the design in that we can optimize a single cell and then configure this cell within the required structure. Each neuron is to be an autonomous entity that will execute weighted summations, thresholding and weight updates in *locked-step*. The detailed neuron architecture is shown in **Figure 3.6**, along with the labelling of a few backpropagation equations as the architecture formulates them. Each neuron contains only a single multiplier and a single adder unit; the figure indicates multiple units for clarity. All signals are multiplexed into the adder and multiplier at specific times so that these devices may

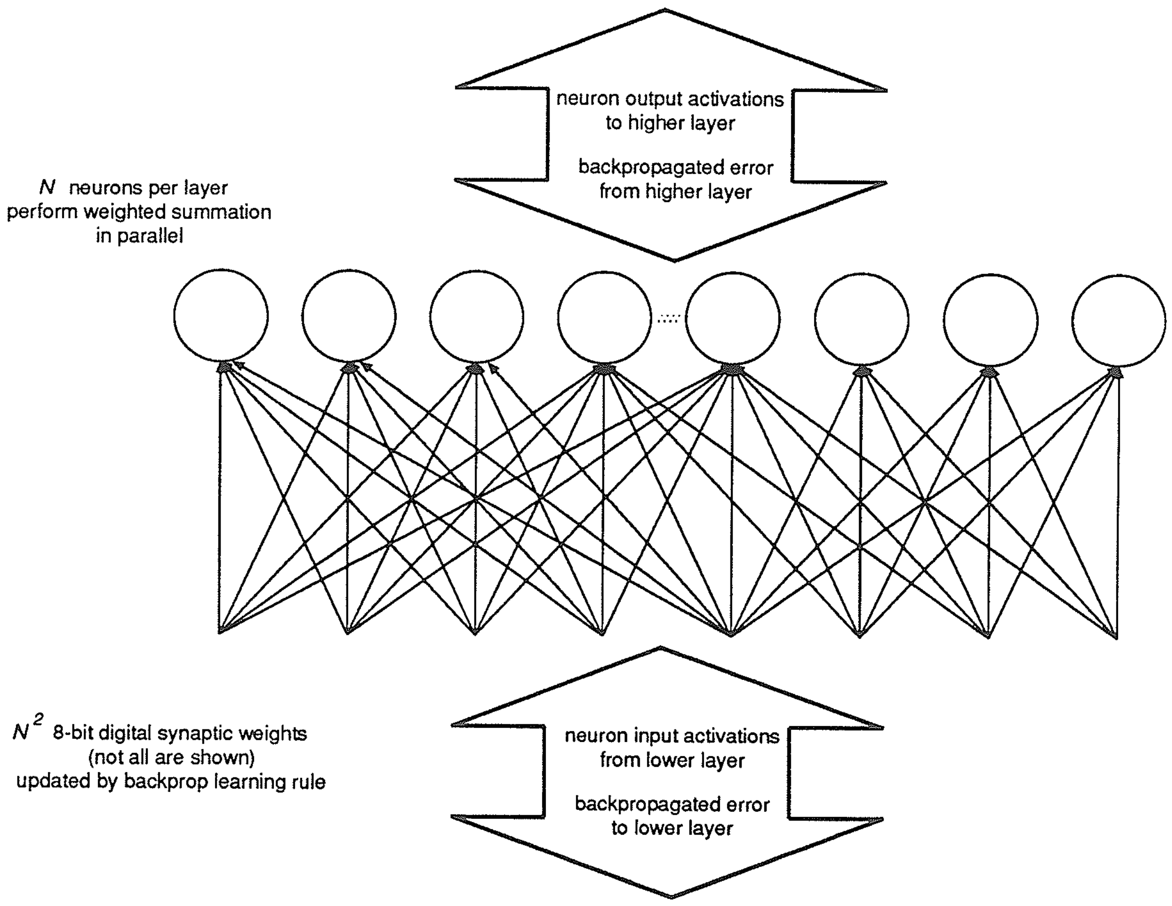


Figure 3.5: **Overview of Backpropagation Layer**

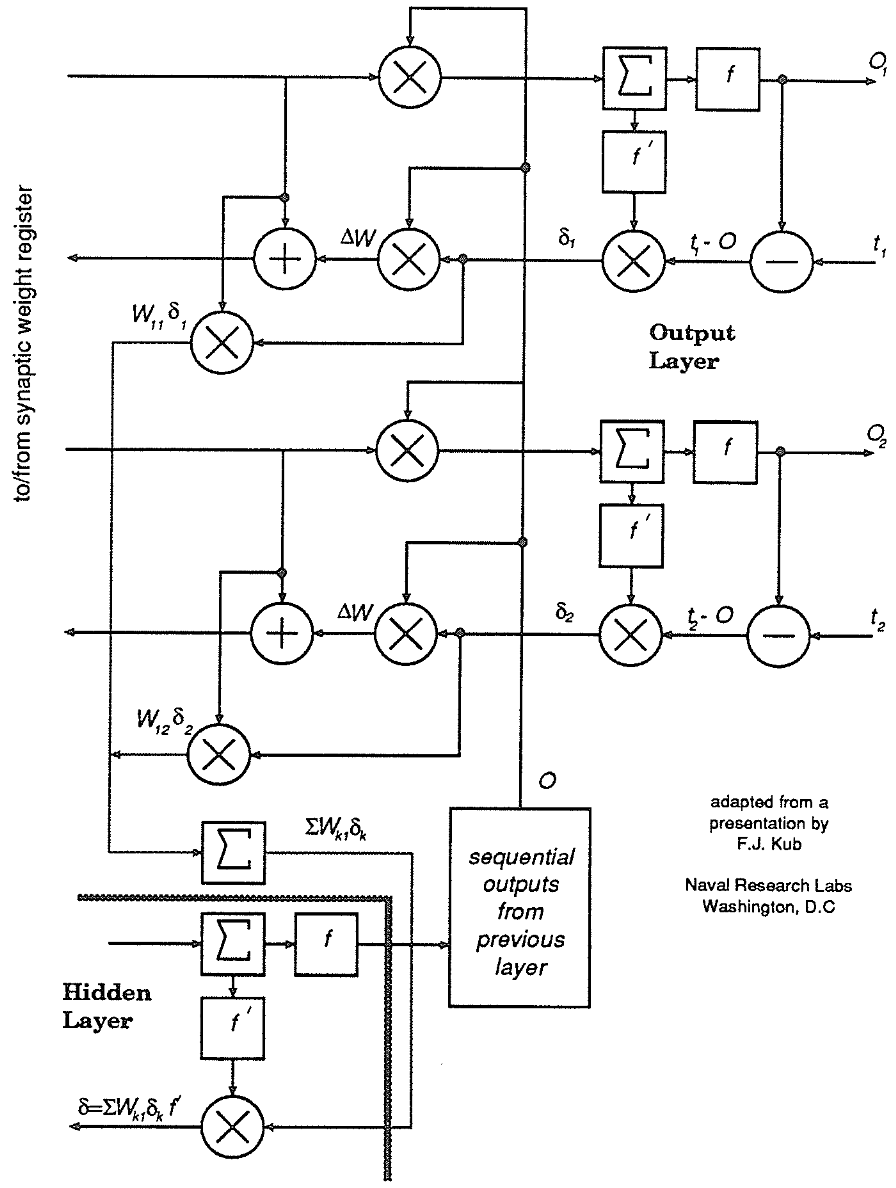

to/from synaptic weight register
ΔW
δ_1
$t_1 - O$
O_1
t_1
$W_{11}\delta_1$
Output Layer
O_2
ΔW
δ_2
$t_2 - O$
t_2
$W_{12}\delta_2$
O
$\Sigma W_{k1}\delta_k$
sequential outputs from previous layer
adapted from a presentation by F.J. Kub
Naval Research Labs Washington, D.C
Hidden Layer
$\delta = \Sigma W_{k1}\delta_k f'$
f
f'

Figure 3.6: **Architecture for Backpropagation Neural Network**

be shared. This is not a disadvantage, since the backpropagation algorithm requires that calculations be performed in sequence. The major devices within the network will be explained in detail in the next few pages.

Examination of the backpropagation equations shows that the most efficient way to implement them is to *broadcast* a *single* input to all neurons. In the forward mode, each neuron accumulates the contribution to its total input by multiplication with its local weight matrix. The inputs coming from each neuron in a lower layer are cycled in sequence. The backward mode requires this same type of multiply and accumulate function. In the backward mode, however, a single error delta is broadcast to all neurons in the previous layer—much the same way as an input is in the forward mode. (Think of the network as operating in reverse). This accumulation requires the distribution of weights orthogonal to the forward mode requirements; hence, local weight storage at each neuron is not possible. We now describe the major components that will perform the multiply-and-accumulate function. That is, the multiplier, adder, sigmoid thresholding and weight storage. We make no attempt to describe the remaining additions and subtractions needed to form the weight updates, etc., as these tasks are trivial and can be accomplished by multiplexing signals into and out of these components as required.

The multiplier module conforms to the bit-serial mode of operation for calculations. **Figure 3.7** shows a basic cell used in the multiplier, while **Figure 3.8** shows how these cells are connected to form an 8-bit two's complement serial multiplier. This pipelined multiplier takes two 8-bit two's complement numbers and produces an 8-bit two's complement result — least significant bit first. As the multiplicand and multiplier bits are *piped* into a multiplier cell, the cell *captures* the proper multiplier

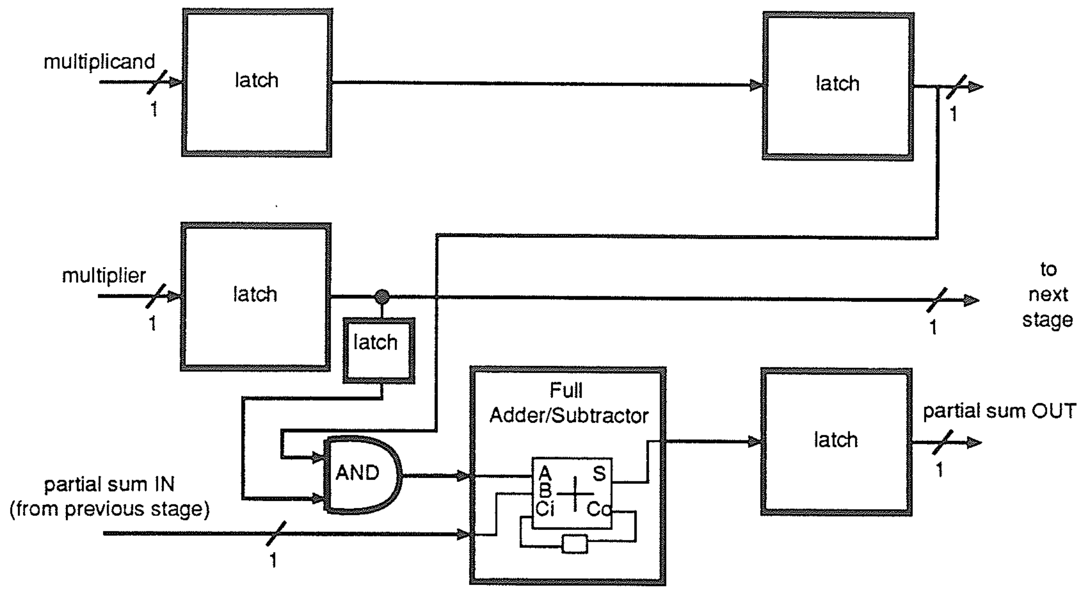


Figure 3.7: **Bit-Serial Pipelined Multiplier Cell**

bit in each stage; i.e. stage 1 captures the least significant multiplier bit, while stage 8 captures the most significant bit. As the multiplicand is piped through two delays to the next stage, and the bit-wise multiplication is formed by the AND gate, the full adder creates the partial sum and pipes it into the next stage. A control signal (not shown) truncates the least significant 8-bits of a 16-bit product to produce the final 8-bit result; this is normal for bit-serial pipelined multipliers. The control signal also resets or sets the carry flip-flop in the full adder section as required by the two's complement algorithm; i.e. the first seven stages are full adders that require a '0' carry bit, while the last stage is a full subtracter that is implemented by using a normal cell with an inverted input and the initial carry bit set to a '1'. Note that we have alleviated the need for sign extension capture logic (a requirement of two's complement arithmetic), as described in [7], by including an XOR gate in the output, described in [8]. This multiplier cell was completely laid out by hand in a

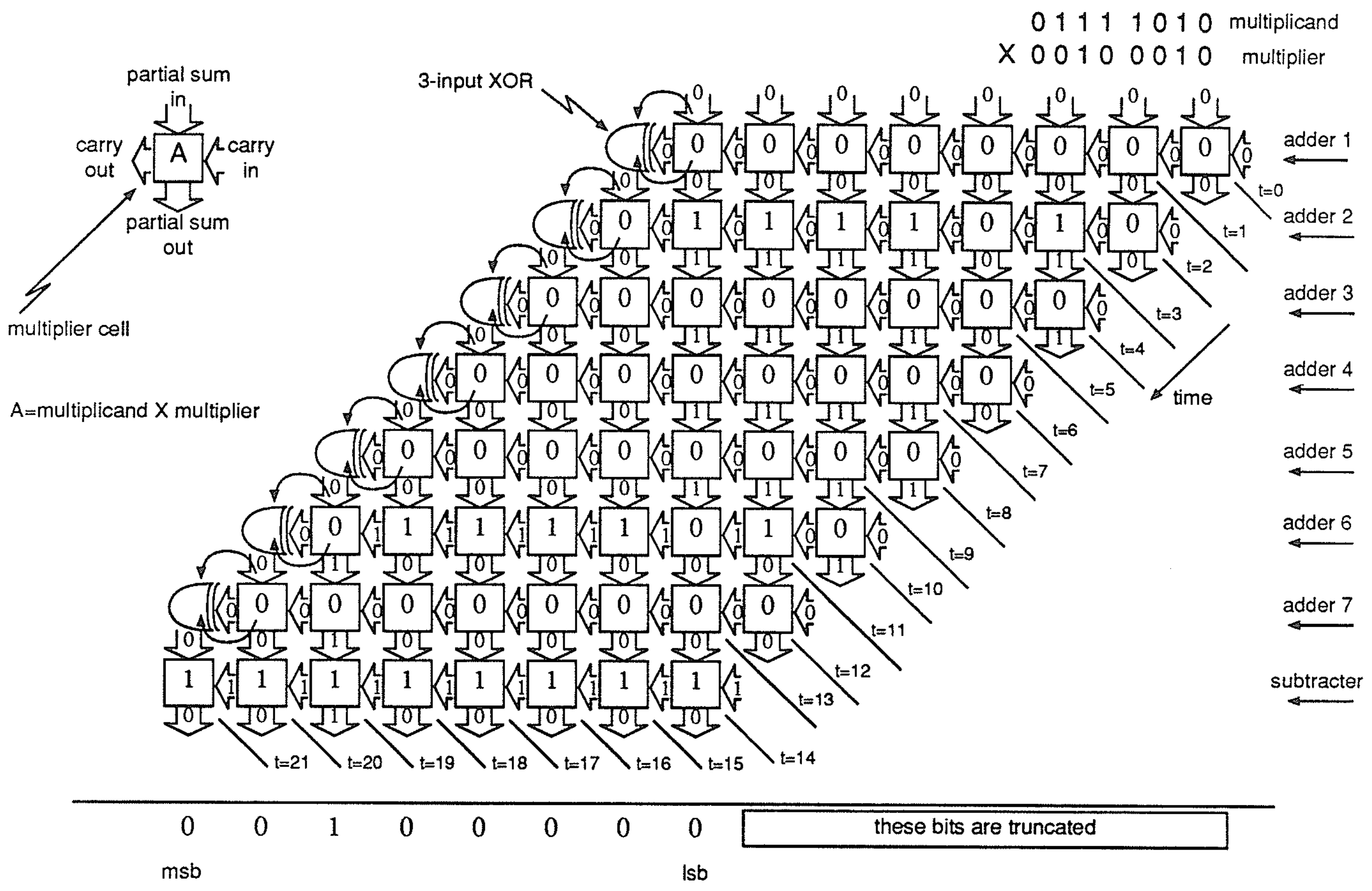


Figure 3.8: **Bit-Serial Pipelined Multiplier** [8]

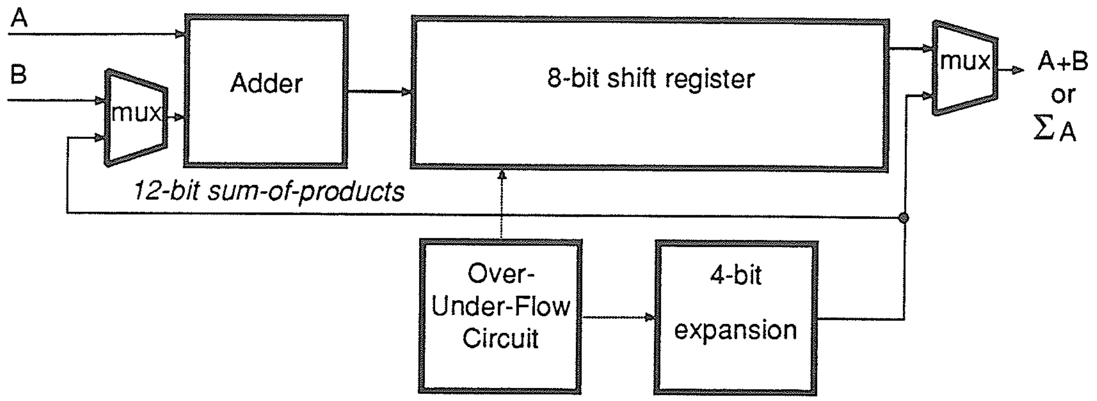


Figure 3.9: **Bit-Serial Pipelined Adder**

custom design using the novel techniques described in the previous section. The total area required by the multiplier is 1458μm $\times 699\mu$m, and the time to perform eight 8-bit multiplications is 80 clock cycles; a 16 clock latency is inherent in the multiplier. Once the latency has passed, new multiplication results come every 8 clock cycles, as new multiplicand and multiplier bits follow their predecessors into the multiplier cells. An example of a bit-serial multiplication using 8-bit two's complement data is also shown in **Figure 3.8**.

The bit-serial adder is also pipelined so that as the 8-bit data from the multiplier is being pumped into it, a single full adder cell can be used to implement a summation of products function, as shown in **Figure 3.9**. The adder cell is an 8-bit two's complement implementation with multiplexed inputs and a 4-bit expansion facility; this facility is necessary to allow the 8-bit summation of products to expand as needed. The expansion is implemented by using a standard up/down counter modified slightly so that the two's complement representation of numbers is calculated *on the fly* and no extra clock cycles are required. Over- and under-flow of data is handled by a circuit that *clamps* the stored value to the limits of two's complement 12-bit accuracy

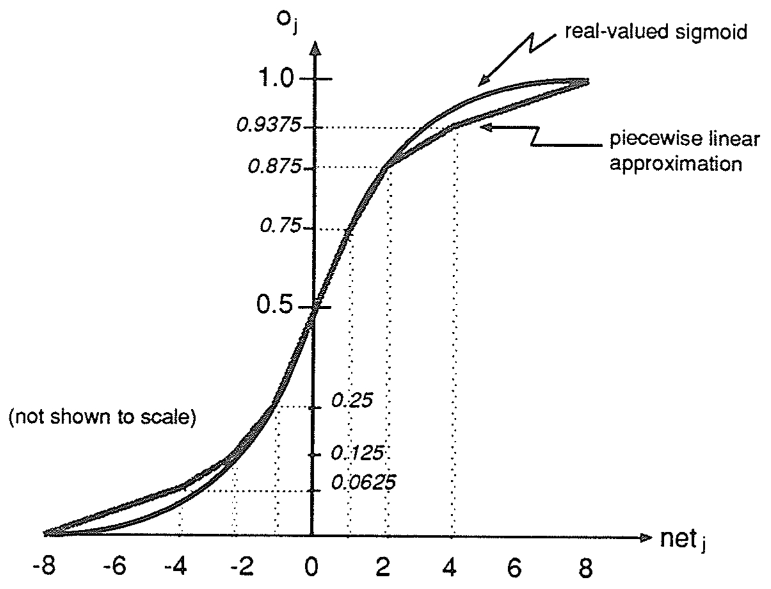
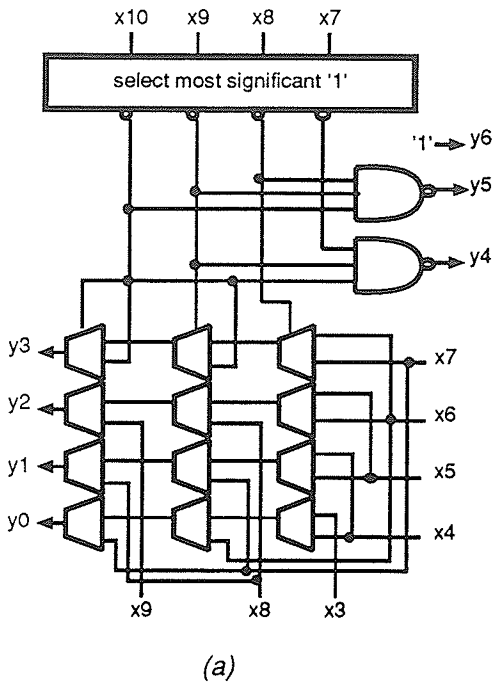


Figure 3.10: **Sigmoid Function: continous and pwl approximation**

(-2048_{10} to $+2047_{10}$) during the weighted summation function. The adder is also used for the execution of general addition as required by the algorithm. For this purpose, the expansion circuitry is disabled and 8-bit accuracy is realized (-256_{10} to $+255_{10}$). The adder module layout was also 'hand-crafted' using the novel VLSI design strategy of the previous section and required approximately 1466μm $\times 375\mu$m of silicon area. The time to perform eight 8-bit additions is 64 clock cycles.

Neural networks require an activation function to limit their outputs to some arbitrary set of values. The backpropagation algorithm uses the sigmoid function, as shown in **Figure 3.10**. The sigmoid maps continuous input values into output values in the space defined by $[0,1]_{10}$. Digital implementations require that the input/output states be quantized to some specific level of accuracy. Our design uses 12-bit accuracy and therefore we introduce a piece-wise-linear (pwl) approximation to the sigmoid,



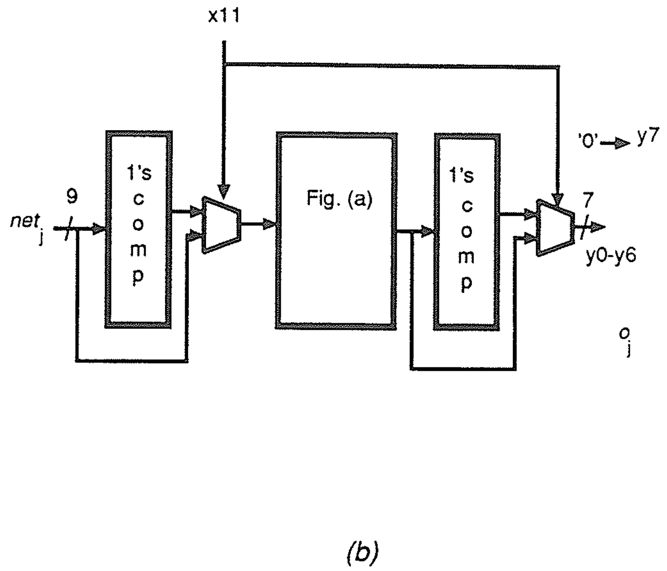


Figure 3.11: **PWL Activation Function: (a) Circuit to Implement Table 3.1 (b) Circuit to Implement Full PWL Activation Functions**

modelled after Myers and Hutchinson [9], and shown in **Figure 3.10**; breakpoints of the curve are also indicated in the figure. The approximation is based on A-law companding for pulse code modulation (PCM) used in digital transmission systems. The sigmoid is implemented by a large array of multiplexors that map the 12-bit input between $[-8,+8]_{10}$, defined by assuming a radix point within the binary data, into an 8-bit output in the range $[0,1]_{10}$. **Figure 3.11(a)** shows the circuit that is used to implement **Table 3.1** (the truth table for positive values of inputs) and **Figure 3.11(b)** shows how the complete sigmoid curve is implemented.

Unlike the pwl sigmoid function described in [9], we have opted to approximate the two's complementing required to realize the *duality* of the positive input values

net_j												o_j							
x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	a	b	c	d	$X^{\dagger}$	X	X	0	1	0	0	a	b	c	d
0	0	0	0	1	a	b	c	d	X	X	X	0	1	0	1	a	b	c	d
0	0	0	1	a	b	c	d	X	X	X	X	0	1	1	0	a	b	c	d
0	0	1	a	b	c	X	X	X	X	X	X	0	1	1	1	0	a	b	c
0	1	a	b	c	X	X	X	X	X	X	X	0	1	1	1	1	a	b	c

† X = don't care

Table 3.1: **PWL Activation Truth Table for Positive Input** [20]

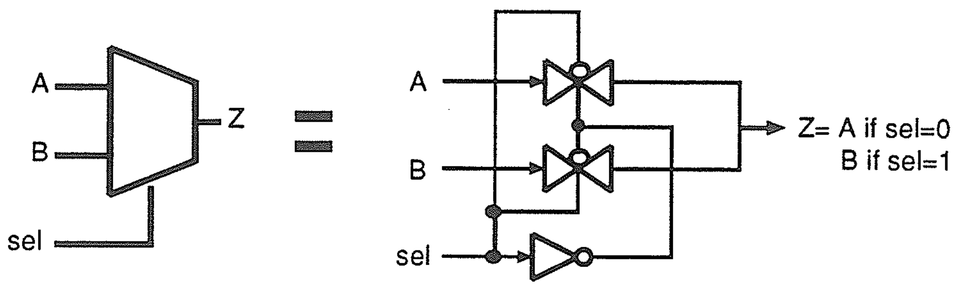


Figure 3.12: **2-Input Active Multiplexor**

by using the one's complement instead; a one's complement is simply an array of inverters, while a two's complement requires the use of an adder circuit. The error introduced by this approximation is negligible and offers us an area savings of 10%. Also, since many 2-input multiplexors are required by this design, we have opted to use an *active* multiplexor that requires no power or ground supplies; a pair of cross-coupled transmission gates as shown in **Figure 3.12**. Since these devices have very little fan out capabilities, drivers were placed on outputs as required. Normal 2-input multiplexors require 3 NAND gates and an invertor; therefore, the active multiplexor saves 70% in silicon area. Again, the VLSI ideas presented in the previous section were used to implement the pwl sigmoid approximation and an area of 422μm $\times$602μm was obtained. The delay time for signals to propagate from input to output fall in the range $[10, 50]_{ns}$.

Each neuron utilizes a single multiplier, adder and sigmoid generator within its own autonomous architecture. This offers the advantage of neuron parallelism and offsets the time required to perform the bit-serial computations of the entire neural network structure.

A single parallel multiplier/adder combination that is multiplexed to update each of the 64 synaptic connections in an eight neuron network would require approximately 192 clock cycles to complete backpropagation learning, whereas eight serial multipliers would need approximately 240 clock cycles for the same computation. However, since bit-serial designs can operate at a much higher clock speed, because of data pipelining, the overall computational speed (expressed as interconnections per second) can be greater. A reasonable assumption would be that the parallel version could work well at 10 MHz while a bit-serial design could operate comfortably at 20 MHz.[5] This implies that the parallel version would operate at approximately 3.3 MiPS (Millions of interconnections Per Second) while the serial version operates at approximately 5 MiPS—an increase in speed of around $1\frac{1}{2}$ times. A fully parallel multiplier that offers similar data resolution to a bit-serial multiplier design would require approximately $3 \times 10^6 \mu\text{m}^2$ of silicon area.

A parallel pipelined structure, whereby parallel additions are latched at each cycle, would allow for clock rates near 20 MHz within an area of $\frac{1}{2}$ of our bit-serial design. However, our bit-serial design has the potential to run at speeds near 40 MHz. This translates into an equivalent throughput of 1.2 MiPS/neuron for both a chip that contains eight serial multipliers and one with 16 parallel pipelined multipliers. **Table 3.2**

[5]The reason for a slower speed in parallel structures is that signals are delayed as they *ripple* through adder stages, while bit-serial pipelining latches signals such that gate delays are minimized.

COMPARISON OF MULTIPLIER ARCHITECTURES			
	fully parallel	parallel-pipelined	bit-serial
multiplier area* ($\times 10^6 \mu m^2$)	3	0.85	1.7
speed, (MiPS)	3.3†	3.3††	5†††
throughput, (MiPS/neuron)	3.3††††	1.2††††	1.2

* 3μm CMOS
† at 10 MHz, 1 multiplier
†† at 10 MHz, 8 multipliers
††† at 20 MHz, 8 multipliers
†††† gate delays dictate the speed of operation

Table 3.2: **Comparison of Multiplier Architectures**

summarizes these results. These values are, however, all empirical and based solely upon typical performance approximations; physical experimentation and observation will provide more qualitative results. The table illustrates the viability of a bit-serial design architecture.

Single-bit data bus structures are attractive in VLSI architectures. A parallel data bus creates a bottleneck for routing strategies. A structure of parallel communicating processors (like a neural network) via parallel data buses further complicates routing. Since we can obtain the same computational throughput with a serial or parallel design structure, serial processing is justifiably attractive. A parallel structure of bit-serial devices with allow for better communication which implies higher densities per chip.

Each neuron also requires the use of shift registers to hold synaptic weight information. This is where the previous section on VLSI design strategies really shines. Usually, designs requiring many DFFs consume huge amounts of silicon area. A $16 \times 17 \times 8$-bit shift register was laid out using the stackable cells described previously and, along with some small multiplexing, required less than 1/2 the area of an oversize

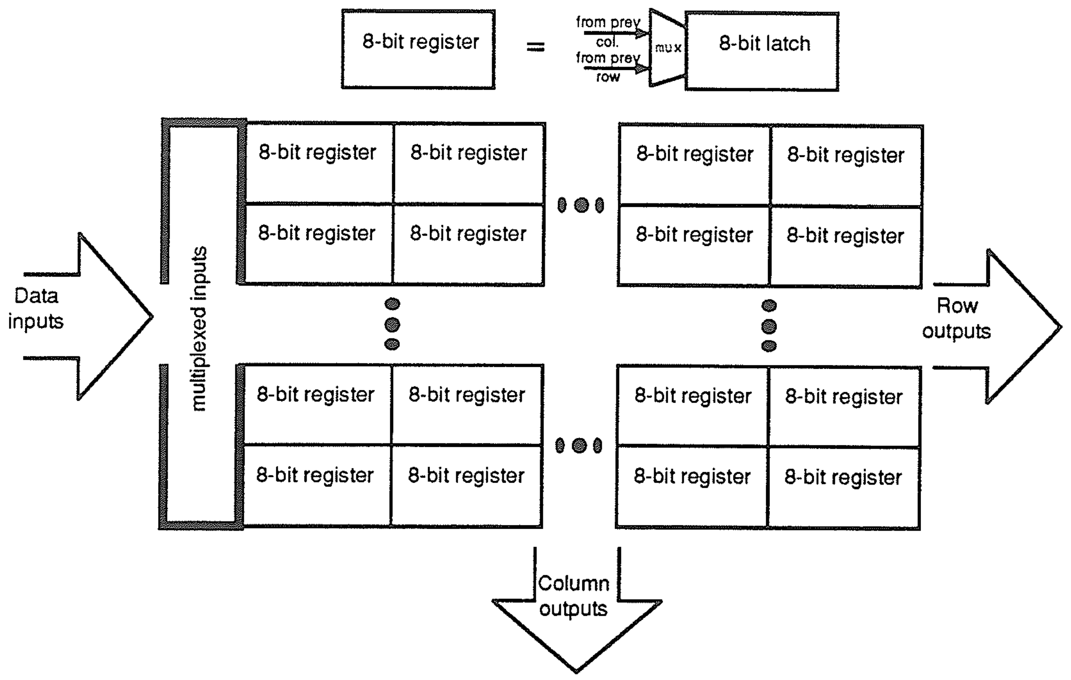


Figure 3.13: **Synaptic Weight Register**

die available from CMC. The area was approximately $6000 \times 3000\mu m$ and the density was approximately 70,000 transistors—a major breakthrough in shift register layout. This translates into a density on the order of 250 μm^2/transistor. This doesn't seem too impressive when you consider that a minimum size p-type transistor alone only requires about 80 μm^2. In comparison, though, the inverter in our standard cell library requires 1620 μm^2 (or 810 μm^2/transistor), therefore, our novel design strategy has given us an equivalent area savings of nearly 70%.

The synaptic weight register can be configured to operate in two modes as suggested by **Figure 3.13**. In one mode, the shift register shifts data in parallel rows between 8-bit registers. The data is *wrapped around* so that the last register's output is directed into the first register's input of the same row. Similarly, for the second mode of operation, the column mode, data is shifted in a columnar manner such that the last 8-bit register feeds the first 8-bit register of the same column. Each row (column) shifts in stepped-sequence to every other row (column) such that a parallel data structure is accomplished. The column mode of operation is used in the forward pass of the backpropagation algorithm, while the row mode is used for the backward pass and pre-loading of the weight registers. The reconfigurability is essential because upon examination of the backpropagation equations we see that the order of weights that are required in the forward summation of the backpropagation algorithm (to create the weighted sum) is different from that required by the backward error computations (in the calculation of error deltas, δ); in fact, the weights are orthogonal to each other. The diagram in **Figure 3.14** will make the reconfiguring concept a little clearer. For clarity, we assume a 4-neuron structure; the forward summation requires $net_1 = o_a w_{11} + o_b w_{21}$ and $net_2 = o_a w_{12} + o_b w_{22}$, while the backward summation is $\delta_a \propto \delta_1 w_{11} + \delta_2 w_{12}$ and $\delta_b \propto \delta_1 w_{21} + \delta_2 w_{22}$. From these equations, it is obvious that broadcasting each input (in sequence) to the neural network in the forward pass of the algorithm, and broadcasting each error derivative in the backward pass requires the use of orthogonal synaptic weights.

The design also includes registers to latch the input data (activations from a previous layer or input vectors), learning rate (η), momentum (α), and the back-propagation error terms (δ). Other modules, such as the multiplier control unit and instruction decoder, are mentioned but not described. An 11-bit hybrid cellular automata (HCA) is included to *seed* the synaptic weight registers with pseudo-random weights.

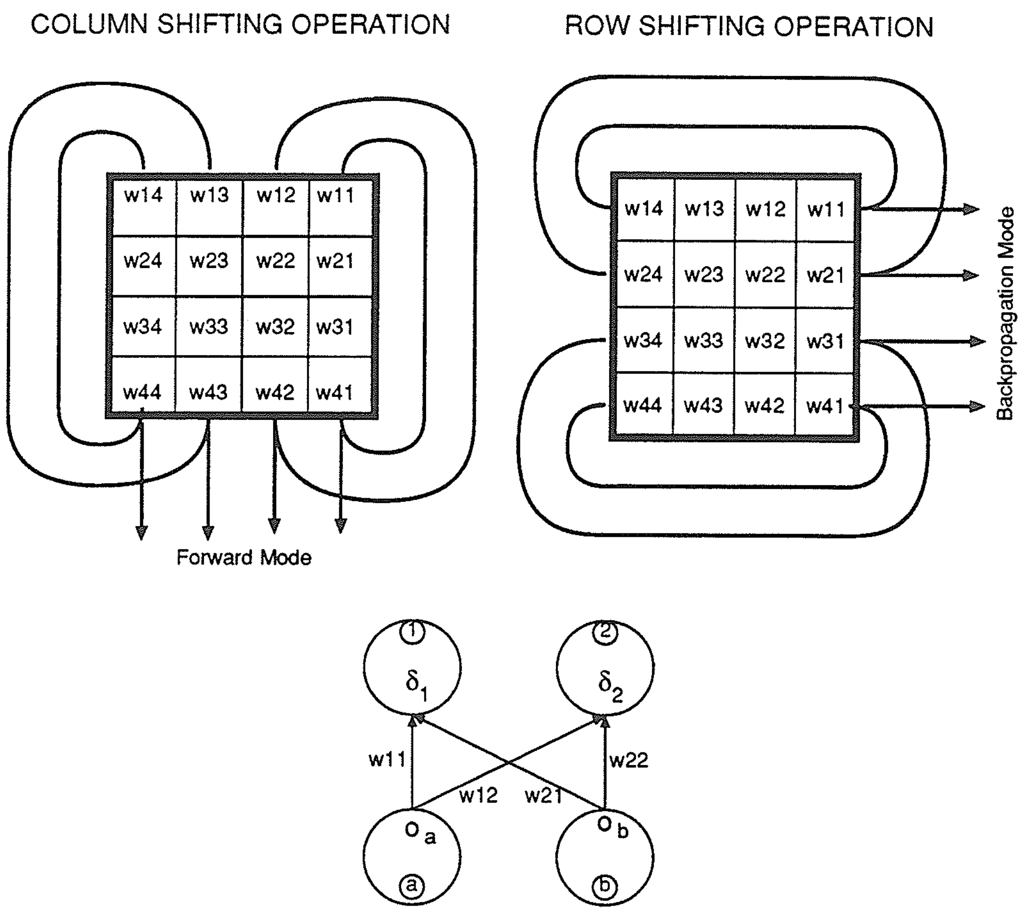


Figure 3.14: **Reconfiguration to Support Backpropagation Learning**

Although the parallel structure of our ANN design offers the same fault tolerance (hardware redundancy) capabilities of a real neural network, i.e. graceful degradation,[6] we have conformed to a bit-serial testing strategy similar to one described by Denyer [82]. The HCA outputs are multiplexed throughout the design to provide testing patterns to most parts of the ASIC. An external tester would have to provide the necessary computations for signature analysis of the resultant patterns to test for faults.

The ASIC implements eight neurons in a single layer of a multi-layer neural network. Weights are stored on-chip and updated by backpropagation. Learning rate and momentum terms are included in the calculations. The neurons operate in a parallel fashion using bit-serial calculations. Each single neuron contains a multiplier, adder, and threshold unit. A single clock is distributed throughout the entire ASIC. An $8 \times 9 \times 8$-bit synaptic weight register is implemented using our 'overlapping' cells for storage of the 64 synaptic and eight bias weights. All data is in 8-bit two's complement form. The architecture is tightly pipelined so that high clock rates are achievable. An on-chip piece-wise-linear approximation to the sigmoid function provides the neuron activation transfer function. The chip can be cascaded so that multi-layered neural networks can be formed.An HCA is used to initially place random weight values into the synaptic registers and also to provide testing patterns to the circuit. The implementation is in fully-digital 3μm double-level-metal CMOS technology. The total die size is $8200\mu\text{m}\times 7600\mu\text{m}$ with approximately 38,000 transistors and 66 pins, and will fit perfectly in an oversize die available from CMC. The power consumption is approximately 1 Watt at 20 MHz.

[6]Graceful degradation refers to the ability of a system to recover from a fault in a graceful manner.

The chip has not been fabricated but is complete and ready for submission, if one so desires. A complete simulation of this ASIC is virtually impossible because of the vast amounts of data that must be processed in a parallel manner on a serial computer; along with tremendous storage capacity requirements. However, simulations have been performed on each module within the ASIC, and all models perform as expected. Simulation of a single neuron indicated that the design was operational. Comparison to speeds of similar architectures was not performed as most research-oriented VLSI neural network designs do not report these findings. The commercial designs described in **Section 2.3** utilize fast DSP chips. These chips have been optimized over years of research and, therefore, comparisons are not included.

Figure 3.15 shows the ASIC design layout in metal-1 only, while **Figure 3.16** shows a breakdown of the chip's structure.

The following is a general sequence of events that the ASIC follows when it is performing learning as controlled by an off-line computer:

1. Initialize weights.
2. Latch inputs, η and α.
3. Forward propagate inputs (multiply and accumulate).
4. Send sigmoid activation off-chip (neuron output).
5. Calculate derivative of output and wait for error input.
6. Calculate error deltas (multiply and accumulate) and send off-chip (backpropagation error).
7. Update weights on-chip.
8. Wait for new input (goto 2 upon receipt of new input).

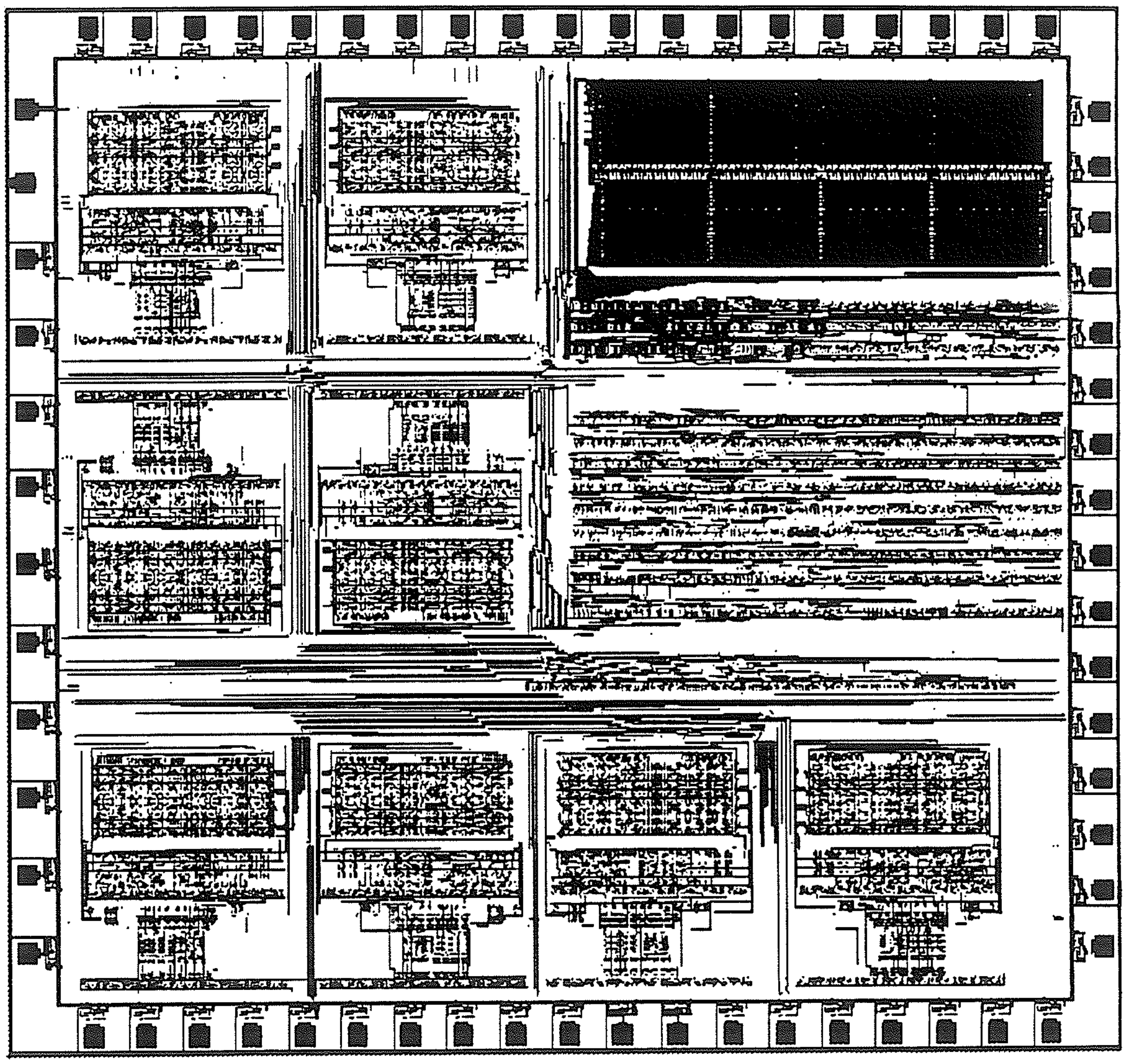


Figure 3.15: **Neural Network with Backpropagation Paradigm ASIC**

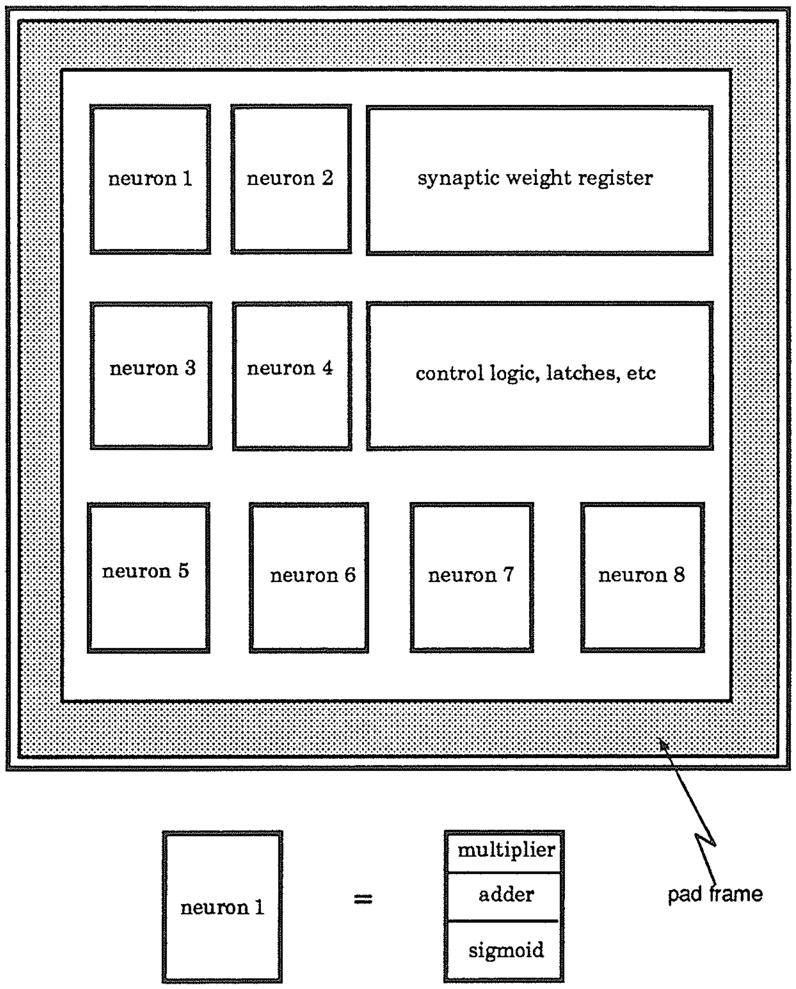


Figure 3.16: **Block-level Diagram of the ASIC Layout**

Technology	3μm CMOS (digital)
Total die size Neuron size Weight storage	$62.32 \times 10^6 \mu m^2$ $2.8 \times 10^6 \mu m^2$ $4.8 \times 10^6 \mu m^2$
# of neurons	8
# of pins: input output test	66 54 10 4
Power	1W @ 20 MHz
# of transistors	38,000
Arithmetic	8-bit bit-serial 2's complement

Table 3.3: **General ASIC Description**

During recall, weight updating and backward error propagation modes are disabled by instructions from the host computer.

Table 3.3 summarizes the implementation of the ASIC. The ASIC layout was accomplished by a hierarchical place and route of the major cells, along with a full custom layout of several modules, as was previously discussed in this section. Edge was used for the complete design, including full- and semi-custom design, as well as schematic capture.

The ASIC is a single layer of a multi-layer backpropagation neural network and can be used as the major component in a hardware accelerator board for simulating this paradigm. The chip can be cascaded to construct a multi-layer network with little additional circuitry. The chip can be controlled by an off-line computer that can act as a user interface to the network, providing control, input and result feedback functions. A typical configuration could be a neural hardware board interfaced to a Sun Microsystems Workstation via VMEbus or SBus, as shown in **Figure 3.17**.

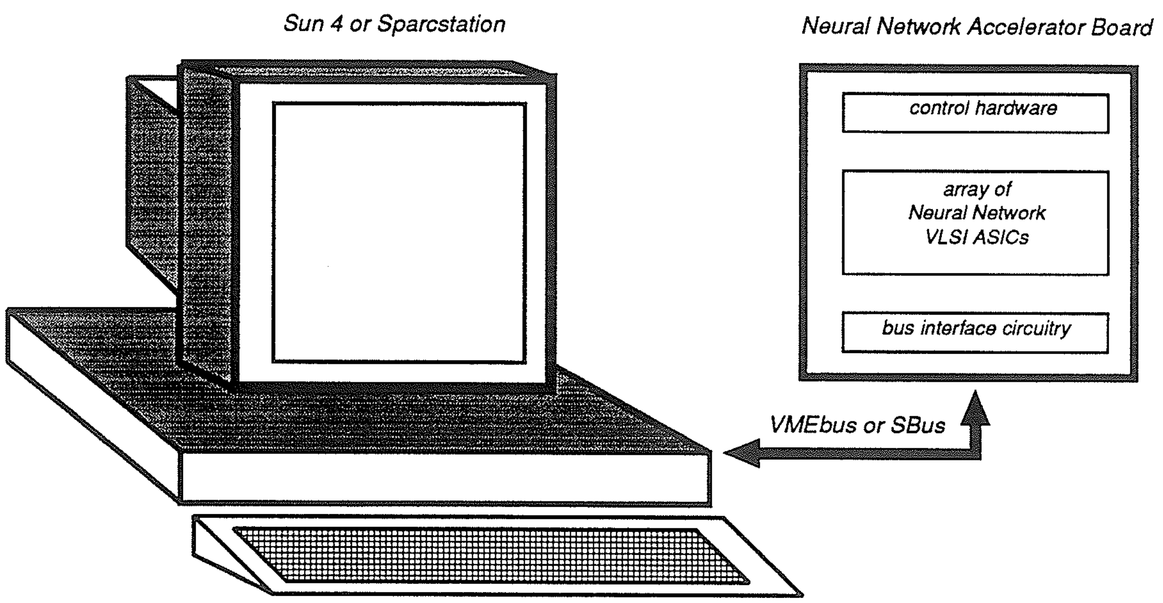


Figure 3.17: **Typical use of the VLSI ASIC in a Neural Network Simulation System**

Chapter 4

Conclusions and Recommendations

We have succeeded in implementing the backpropagation learning paradigm in a VLSI ASIC. The digital architecture in our neural network displayed high computational throughput. The density of neurons that could be placed on the chip was very disappointing. The inclusion of on-chip learning did not affect the neuron density; the multiplier was the major consumer of silicon area.

We recommend the use of a new multiplier policy. The bit-serial approach seemed like a good design choice throughout this project's development. However, large amounts of silicon were still consumed by the multiplier. The use of single multiplexed multipliers should be investigated further, as should parallel multiplier structures.

This implementation of a neural network proved that digital designs of neural networks should continue to be investigated. The viability of digital neural network designs is evidenced by the tremendous amount of architectures reported on herein. We recommend that the use of stochastic processing be further analyzed; especially the work of Tomlinson et al. [69]. These processes are good candidates for the use of

HCAs. Further, the densities achieved by these designs are very impressive.

A software environment was attempted, but was not seriously considered. The ASIC is intended for use by a serial computer as a hardware neural accelerator. The task of developing a neural board and the development of a software interface should also be attempted.

The ASIC can be cascaded to form any number of layers of a multi-layer feed-forward network. We recommend the extension of this design to sequential networks and single layer feedback networks. An extension to allow chips to be placed side-by-side to form arbitrarily long layers of neurons should be investigated. The chip as it stands now, cannot be configured for larger numbers of neurons per layer and therefore would have to be redesigned.

Alternative forms of storage should also be considered. The storage of weights can escalate tremendously as larger designs are established. Our design was small enough that storage was not a problem. Static and dynamic RAM structures with multiport access capabilities should be considered.

Backpropagation learning can be very slow, even with an accelerator chip as we have developed. An exploration into the many possibilities of backpropagation speedup techniques would be beneficial.

We have also created a novel VLSI design layout strategy that could form the basis for a unique library of standard cells. These cells could be used to create high density layouts of regular structures.

Appendix A

Generalized Delta Rule

This appendix will attempt to summarize the inception of the backpropagation of errors, or backprop learning paradigm from its roots in the Perceptron Convergence Theorem, through the least mean square association, to its most basic form in the Generalized Delta Rule. We do not include any backprop speedup techniques here (references to these are made elsewhere)—we simply want to show, from a purely mathematical perspective, how the original version of the backpropagation learning paradigm (sometimes referred to as vanilla backpropagation) was developed. The derivative is a summary of McClelland and Rumelharts Exploration in PDP Handbook (Chapter 5) [86], along with Rumelhart, Hinton, and Williams' Learning internal representations by error propagation, PDP vol. 1, pp. 318-362 [2]. There are several ways to develop the actual algorithm, but we will only present one possible way—as described in the references stated. Other possible derivations can be found in Rumelhart, Hinton, and Williams' Learning internal representations by backpropagating errors, Nature 323: 533-536 (1986) [86], and an interesting statistical comparison in Neural Network Learning and Statistics by Halbert White, AI Expert Magazine, Dec. 89, pp. 48-52 [30]. From this it is interesting to note that backpropagation is actually an application of neural network learning of statistical methods proposed by Herbert Robbins and Sutton Munro in 1951. Nevertheless, what follows is a strictly mathematical derivation of the backpropagation rule for those interested in the mathematics behind those infamous *hidden units.*

We start off by stating that the backpropagation learning rule is a complete generalization of the Widrow-Hoff error correction rule, sometimes referred to as the least-mean-square (LMS) learning rule or the delta rule (in reference to the differences in target and output values that create error derivatives). It can also be shown that the delta rule is essentially the same rule used in the perceptron.

The perceptron neural network used a simple linear threshold activation function. The networks were usually confined to single layers of neurons and binary input/output values. That is, the neuron summation was given by

$$net = \sum_i w_i i_i \tag{A.1}$$

and the activation was given by

$$o = \begin{cases} 1 & \text{if } net > \theta \\ 0 & \text{otherwise,} \end{cases} \tag{A.2}$$

where i_i is the input to the neuron, w_i is the corresponding synaptic weight, o is the neuron output, and θ is the threshold over which the activation function becomes active. The change in the threshold, $\Delta\theta$, is given by

$$\Delta\theta = -(t_p - o_p) = -\delta_p \tag{A.3}$$

where p is the presentation pattern index, t_p is the target value for a specific pattern, o_p is the output generated by the net for pattern p, and δ_p symbolizes the difference between desired and actual output. Finally, the change in the weights, Δw_i, are given by

$$\Delta w_i = (t_p - o_p)i_{pi} = \delta_p i_{pi} \tag{A.4}$$

where i_{pi} indicates the input to the neuron for pattern p. If *net* is greater than θ, the unit is turned on; otherwise it is turned off. The output vector is compared with the desired result (a similar vector if classification is performed or a more complex vector for mapping) and the threshold is either incremented or decremented by '1'. This is similar for the change in weight, i.e. binary changes only. The relative simplicity of this procedure is contrasted by the guarantee that a set of weights that correctly classifies the input vector will be found **if such a set of weights exists**—the so-called perception convergence theory. Also, an appropriate mapping of input to output vectors will be found **if such a mapping exists.** Unfortunately, this mapping does not always exist and the perceptron algorithm is thus flawed. Minsky and Papert, in Perceptrons (1969) stated a simple example of the limitations of the perceptron by showing that it cannot compute the exclusive-or (XOR) function. The XOR problem is found in many examples when the problem is reduced to lower classification levels. The class of problems that can be solved by a perceptron happens to be linearly separable functions. The perceptron can be made to solve the XOR problem in one of two ways: (1) by adding a third dimension (variable) to the problem; and, (2) by allowing a second layer (multi-layer) of perceptron units. This new form of multi-layer perceptrons defined the classes **input**, **hidden**, and **output units**. The input units receive input patterns, while the output units have corresponding target patterns—these are the external connections to the outside world. The hidden units receive their inputs from units in lower layers (in the case of feed forward nets), or from lower, similar and/or higher layers (in feedback networks). The original perceptron learning algorithm did not account for multiple layers and therefore could not be applied to teach these networks.

Another major single-layer learning algorithm being developed was the LMS learning procedure. This procedure makes use of the delta rule for adjusting connection strengths[1]. The term LMS is used to stress the fact that the learning rule tries to minimize an error measure in its performance. In this learning procedure, purely linear output activations are used such that the output of units is given by

$$o_j = \sum_i w_{ij} i_i \tag{A.5}$$

where i and j are unit indices, o_j is the output of the j^{th} unit, i_i is the input from i^{th} unit, and w_{ij} is the connection strength from unit i to unit j. The error function is the summed square error defined to be

$$E = \sum_p E_p = \sum_p \sum_j (t_{pj} - o_{pj})^2 \tag{A.6}$$

where p is the input pattern index for the training set, j is the output unit index, and E_p is the error for pattern p. The target value, t_{pj}, is the desired output for the j^{th} output unit during the presentation of pattern p, and o_{pj} is the actual output of unit j during this same pattern input. The basic algorithm is to find a set of weights, w_{ij}, that will minimize the function E. The LMS procedure finds these weights by a method called **gradient descent**. The idea of gradient descent is to change the weight matrix such that a change in weight proportional to the negative of the derivative of the error for the current pattern for each weight will decrease the

[1]Note that the only difference between LMS and perceptrons is that LMS uses outputs with continuous values.

overall error to a global minimum, by the equation

$$\Delta w_{ij} = -k\frac{\partial E_p}{\partial w_{ij}} \tag{A.7}$$

and substituting for the derivative in E we get

$$\Delta w_{ij} = \eta \delta_{pj} i_{pi} \tag{A.8}$$

where η is a constant of proportionality, $\delta_{pj} = t_{pj} - o_{pj}$ is the difference between target and actual output values for unit j during pattern p, and i_{pi} is the input from unit i during this same pattern. Notice that this learning rule is exactly like that of the perceptron as was previously stated—the only difference is the perceptron's use of binary (i.e. '1' and '0') values. This LMS learning rule was developed completely independent of another rule, the delta rule; however, it turns out that it is the identical result—hence the constant interchanging of terminology for this particular rule.

Although the LMS procedure is useful in certain respects, it turns out that any linear system of activation functions cannot compute more in multiple layers than they can in a single layer. Therefore, like the perceptron, LMS still cannot solve the XOR problem. The backpropagation of error algorithm was developed to overcome this limitation by combining non-linear perceptron-like units with the LMS error function and gradient descent. We start off our derivation of backpropagation, or the generalized delta rule, by presenting the rule for changing connection weights following an input/output training pair p

$$\Delta_p w_{ij} = \eta(t_{pj} - o_{pj})i_{pi} = \delta_{pj} i_{pi} \tag{A.9}$$

where t_{pj} is the target output, o_{pj} is the actual output during pattern p for output unit j, i_{pi} is the input value from unit i during this same pattern presentation, η is a constant of proportionality, $\delta_{pj} = t_{pj} - o_{pj}$, and $\Delta_p w_{ij}$ is the change made to the connection weight from the i^{th} to j^{th} unit following training pattern p. This is exactly the same as the LMS (delta) training rule and in fact it is just restated. A formal proof of the generalized delta rule will now follow.

If the error function for a neural network upon presentation of a training input/output pair, p, is

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \tag{A.10}$$

and

$$E = \sum_p E_p \tag{A.11}$$

where all variables are the same as before. The delta rule states that we must take the derivative of the error function with respect to the weight. By applying the chain rule we get

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ij}} \tag{A.12}$$

The first part of the equation tells us how the error changes with the output of the j^{th} neuron and the second part tells how much changing w_{ij} will change that output. These partial derivatives are easy to compute and the first half of **Equation A.12**, from **Equation A.10**, is given by

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj} \tag{A.13}$$

that is, the contribution of the j^{th} unit to the error is proportional to δ_{pj}. Since we are now only considering linear output activations,

$$o_{pj} = \sum_i w_{ij} i_{pi} \tag{A.14}$$

from which we get the second half of **Equation A.12** to be

$$\frac{\partial o_{pj}}{\partial w_{ij}} = i_{pi} \tag{A.15}$$

Thus, substituting **Equation A.15** and **Equation A.13** into **Equation A.12** we get

$$\frac{-\partial E_p}{\partial w_{ij}} = \delta_{pj} i_{pi} \tag{A.16}$$

that is proportional to $\Delta_p w_{ij}$ as prescribed by the delta rule in **Equation A.9** where η would be the constant of proportionality.[2] Now if we combine **Equation A.16** with the observation that

$$\frac{\partial E}{\partial w_{ij}} = \sum_p \frac{\partial E_p}{\partial w_{ij}} \tag{A.17}$$

we get that the net change in w_{ij} after one complete cycle of training pattern presentations is proportional to this derivative and hence this delta rule implements a gradient descent in E. By changing the weights after each pattern is presented, we depart from a true gradient descent in E, but provided that the constant of proportionality, η (further referred to as the *learning rate*), is relatively small, the error introduced by this departure will be negligible and thus the delta rule will implement a very close approximation to the error derivative. In fact, with a small learning rate, this rule will find a set of weights minimizing this error function.

We have just shown how the standard delta rule implements gradient descent for linear activations. So far, we have considered networks with no hidden units and the delta rule guarantees an optimum solution set of weights. If we now add hidden units to the network, the derivatives are not so obvious and there is the possibility of several local minima. This derivation will be confined to **multi-layered feedforward** networks only, configured such that units on a lower layer can send outputs to higher layers only and must receive inputs from lower layers—notice that the out-

[2]Note that we have used a different error function here as compared to LMS; however, it can be shown that the two functions offer the same mean-squared error derivative.

puts may also skip layers. We will also generalize our derivation by assuming units with semi-linear activation functions only, since linear activation functions provide no advantages for multi-layer networks. A semi-linear function is one in which the activation function is a nondecreasing and differentiable function such that

$$net_{pj} = \sum_i w_{ij} o_{pi} \tag{A.18}$$

where $o_{pi} = i_{pi}$ if the ith unit is an input unit. A semi-linear activation function is given by

$$o_{pj} = f_j(net_{pj}) \tag{A.19}$$

To obtain the generalized delta rule we again take the derivative of the error function with respect to the connection weight, **Equation A.12**, but this time using the semi-linear activation function of **Equation A.19** and we find

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ij}} \tag{A.20}$$

From **Equation A.18** we get

$$\frac{\partial net_{pj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k w_{kj} o_{pk} = o_{pi} \tag{A.21}$$

If we define

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} \tag{A.22}$$

we can now rewrite **Equation A.20** as

$$-\frac{\partial E_p}{\partial w_{ij}} = \delta_{pj} o_{pi} \tag{A.23}$$

This is the similar result as we obtained for the standard delta rule, as in **Equation A.16**, that states that

$$\Delta w_{ij} = \eta \delta_{pj} o_{pi} \tag{A.24}$$

To compute the δ_{pj} in **Equation A.22** we can write

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{pj}} \tag{A.25}$$

The second factor of **Equation A.25** is simply the derivative of the semi-linear activation function f_j for the j^{th} unit evaluated at the net input net_{pj} to that unit, and is given by

$$\frac{\partial o_{pj}}{\partial net_{pj}} = f_j'(net_{pj}) \tag{A.26}$$

Assuming that the j^{th} unit is an output unit, we get, from the definition of E_p, that

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) \tag{A.27}$$

which is the exact result obtained for the standard delta rule as seen in **Equation A.13**. Substituting **Equation A.26** and **Equation A.27** into **Equation A.25** we get

$$\delta_{pj} = (t_{pj} - o_{pj}) f_j'(net_{pj}) \tag{A.28}$$

for the j^{th} output unit. If the j^{th} unit is not an output unit, we can use the chain rule to write

$$\begin{aligned}\sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial o_{pj}} &= \sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ik} o_{pi} \\ &= \sum_k \frac{\partial E_p}{\partial net_{pk}} w_{ik} = -\sum_k \delta_{pk} w_{kj}\end{aligned} \tag{A.29}$$

Again, substituting into **Equation A.25** we get

$$\delta_{pj} = f_j'(net_{pj}) \sum_k \delta_{pk} w_{kj} \tag{A.30}$$

whenever the j^{th} unit is not an output unit. This is a recursive procedure for computing the δ's for all units in a network. Weight changes are then computed according to **Equation A.24**.

We can now summarize from the results just obtained and formally state the generalized delta rule for a feedforward layered neural network with a semi-linear activation function. During the first phase of the rule, the input pattern is presented and propagated forward through the network to compute the output value o_{pj} for each unit. The target pattern is then presented to these output units and an error signal δ_{pj} is produced for each output unit. The second phase involves a backward propagation of the calculated error signal through the network and the appropriate weight changes are executed. The second phase allows the recursive computation of the δ's as shown below. The following three equations summarize the generalized delta rule:

$$\Delta_p w_{ij} = \eta \delta_{pj} o_{pi} \tag{A.31}$$

$$\delta_{pj} = (t_{pj} - o_{pj}) f_j'(net_{pj}), \qquad \text{output unit} \tag{A.32}$$

$$\delta_{pj} = f_j'(net_{pj}) \sum_k \delta_{pk} w_{kj}, \qquad \text{non-output unit} \tag{A.33}$$

Deciding on a semi-linear activation function specifies the exact form of the above equations. A useful activation function for which the derivative exists (a semi-linear function) is the logistic, or sigmoid, activation function given by

$$o_{pj} = \frac{1}{1 + e^{-(\sum_i w_{ij} o_{pi} + \theta_j)}} \tag{A.34}$$

where θ_j is a bias input similar to the threshold in a perceptron. We have that the total input $net_{pj} = \sum_i w_{ij} o_{pi} + \theta_j$ can give us the derivative

$$\frac{\partial o_{pj}}{\partial net_{pj}} = o_{pj}(1 - o_{pj}) \tag{A.35}$$

and therefore, the following three equations for the generalized delta rule for a multi-layered feedforward neural network with a sigmoid activation function are:

$$\Delta w_{ij}(\mathrm{n}{+}1) = \eta \delta_{pj} o_{pi} + \alpha \Delta w_{ij}(\mathrm{n}) \tag{A.36}$$

$$\delta_{pj} = (t_{pj} - o_{pj}) o_{pj} (1 - o_{pj}), \qquad \text{output unit} \tag{A.37}$$

$$\delta_{pj} = o_{pj}(1 - o_{pj}) \sum_k \delta_{pk} w_{jk}, \qquad \text{non-output unit} \tag{A.38}$$

where, in **Equation A.36**, a **momentum** term, α, has been added to increase the learning rate and n indexes the training step. Notice that the bias term, θ_j, in net_{pj} can simply be regarded as another connection weight with a constant input value of '1' and can be modified similar to the other weights. Careful examination of the equations is necessary since the layer indexes are very important; we have applied the convention that letters of the alphabet that occur sooner are considered as indexes for lower layers. That is, layer i is lower than layer j and therefore a unit in layer i will feed signals forward to layer j. In the backward mode, error signals are sent from higher to lower layers. Notice also that the calculation of errors derivatives, δ's, require the availability of the outputs from units in a lower layer and the corresponding weighted connection strengths. Finally, note that the weights are not modified until **after** the error derivatives have been calculated for **all** of the layers. Also note that initial weights must be randomized to break the problem of symmetry.

Bibliography

[1] Minsky, M., and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA: The MIT Press, 1969.

[2] Rumelhart, D.E., and J.L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition.* Vol. 1, *Foundations.* Cambridge, MA: The MIT Press, 1986.

[3] McClelland, J.L., and D.E. Rumelhart. *Parallel Distributed Processing: Explorations in the Microstructures of Cognition.* Vol. 2, *Psychological & Biological Models.* Cambridge, MA: The MIT Press, 1986.

[4] Touretzky, David S., and Dean A. Pomerleau. "What's Hidden in the Hidden Layers?" *BYTE*, August 1989, pp. 227-233.

[5] Butler, Zoe F.; Murray, Alan F.; and Anthony V.W. Smith. "VLSI Bit-serial Neural Networks." *VLSI for Artificial Intelligence*, eds. J. Delgado-Frias and W.R. Moore, 1989, Kluwer Academic, pp. F2/1-F2/10.

[6] Murray, Alan F.; Smith, Anthony V.W.; and Zoe F. Butler. "Bit Serial Neural Networks." *IEEE Conference on Neural Information Processing Systems—Natural and Synthetic*, Denver, 1987, pp. 573-583.

[7] Lyon, R.F. "Two's Complement Pipeline Multipliers." *IEEE Transactions on Communications*, (April 1976): pp. 418-425.

[8] Murray, Alan F., and Peter B. Denyer. "A CMOS Design Strategy for Bit-Serial Signal Processing." *IEEE Journal of Solid-State Circuits*, Vol. sc-20 No. 3 (June 1985): pp. 746-753.

[9] Myers, D.J., and R.A. Hutchinson. "Efficient Implementation of Piecewise Linear Activation Function for Digital VLSI Neural Networks." *Electronics Letters*, Vol. 25 No. 24 (November 23, 1989): pp. 1662-1663.

[10] Heinbuch, Dennis V., ed. *CMOS3 Cell Library.* Reading, MA: Addison-Wesley Publishing Company, 1988.

[11] Weste, Neil, and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Reading, MA: Addison-Wesley Publishing Company, 1985.

[12] Myers, David J., and Peter A. Ivey. "A Design Style for VLSI CMOS." *Journal of Solid-State Circuits*, Vol. sc-20 No. 3 (June 1985): pp. 741-745.

[13] Oklobdzija, Vojin G., and Robert K. Montoye. "Design Performance Trade-Offs in CMOS Domino Logic." *IEEE 1985 Custom Integrated Circuits Conference*, pp. 334-337.

[14] Cadence Design Systems, Inc., *Edge Design Manuals*, version 2.1, USA, 1989.

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

[16] Hebb, D.O. *The Organization of Behavior.* New York: John Wiley & Sons, 1949.

[17] Rosenblatt, F. *Principles of Neurodynamics.* New York: Spartan Books, 1959.

[18] Widrow, B., and M.E. Hoff. "Adaptive Switching Circuits." *1960 IRE WESCON Convention Records, Part 4*, August 1960, pp. 96-104.

[19] Levine, Daniel S. "the Third Wave in Neural Networks." *AI EXPERT*, December 1989, pp. 27-33.

[20] Treleaven, Phillip; Pacheco, Marco; and Marley Vellasco. "VLSI Architectures for Neural Networks." *IEEE MICRO*, December 1989, pp. 8-27.

[21] Lawrence, Jeannette. "Untangling Neural Networks." *Dr. Dobb's Journal*, April 1990, pp. 38-44.

[22] Illingworth, William T. "Beginners Guide to Neural Networks." *IEEE AES Magazine*, September 1989, pp. 44-49.

[23] Hinton, Geoffrey E. "Connectionist Learning Procedures." *Journal of Artificial Intelligence*, Vol.40, No.1-3, September 1989, pp. 185-234.

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

[25] Carpenter, Gail A., and Stephen Grossberg. "The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network." *Computer*, March 1988, pp. 77-88.

[26] Linsker, Ralph. "Self-Organization in a Perceptual Network." *Computer*, March 1988, pp. 105-117.

[27] Hecht-Nielsen, Robert. "Counterpropagation Networks." *IEEE First International Conference on Neural Networks*, San Diego, June 21-24, 1987. Vol. II

[28] Parker, David B. *Learning-Logic.* Center for Computational Research in Economics and Management Science, MIT, April, 1985. Technical Report TR-47.

[29] Werbos, Paul J. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences." Ph.D. Thesis, Applied Mathematics, Harvard University, November 1974.

[30] White, Halbert. "Neural Network Learning and Statistics." *AI EXPERT*, December 1989, pp. 48-52.

[31] LeCun, Yann. *A Theoretical Framework for Back-Propagation.* Toronto: Connectionist Research Group, University of Toronto, September 1988. Technical Report CRG-TR-88-6.

[32] Bryson, A.E., and Yu-Chi Ho. *Applied Optimal Control.* Blaisdell, NY, 1969.

[33] Hecht-Nielsen, Robert. "Theory of the Backpropagation Neural Network." *International Joint Conference on Neural Networks*, Washington, DC, June 18-22, 1989. Vol. I, pp. 593-605.

[34] Parker, D. "Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning." *Proceedings of the IEEE First Annual Conference on Neural Networks*, June 1987. Vol. II, pp. 593-600.

[35] Cailton, J.G.; Angéniol, B.; and E. Markade. "Constrained Back-propagation." *Abstracts of the 1st Annual INNS Meeting, (Special Supplement Issue)*, Boston, Sept. 6-10, 1988. Pergamon Press, NY. p. 539.

[36] Samad, Tariq. "Back Propagation is Significantly Faster if the Expected Value of the Source Unit is Used for Update." *Abstracts of the 1st Annual INNS Meeting, (Special Supplement Issue)*, Boston, Sept. 6-10, 1988. Pergamon Press, NY. p. 216.

[37] Mead, C.A. *Analog VLSI and Neural Systems*, Reading, MA: Addison-Wesley Publishing Company, 1989.

[38] Lee, Bang W.; Lee, Ji-Chien; and Bing J. Sheu. "VLSI Image Processors Using Analog Programmable Synapses and Neurons." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 575-580

[39] Schwartz, Daniel B.; Howard, Richard E.; and Wayne E. Hubbard. "A Programmable Analog Neural Network Chip." *IEEE Journal of Solid-State Circuits*, Vol. 24 No. 2, (April 1989): pp. 313-319.

[40] Furman, B.; White, J.; and A.A. Abidi. "CMOS Analog IC Implementing the Back Propagation Algorithm." *First Ann. INNS Mtg.*, September 6-10, 1988, Boston, MA. Abstract also publ. in *Neural Networks* **1**, Supp. 1 (1988), p. 381.

[41] Fisher, W.A.; Fujimoto, R.J.; and M.M. Okamura. "The Lockheed Programmable Analog Neural Network Processor," *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 563-568.

[42] Goser, Karl; Hilleringmann, Ulrich; Rueckert, Ulrich; and Klaus Schumacher. "VLSI Technologies for Artificial Neural Networks." *IEEE MICRO*, December 1989, pp. 28-44.

[43] Caviglia, Daniele D.; Valle, Maurizio; and Giacomo M. Bisio. "Effects of Weight Discretization on the Back Propagation Learning Method: Algorithm Design and Hardware Realization," *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 631-637.

[44] Kub, F.J.; Ancona, M.G.; Mack, I.A.; Moon, K.; and C.T. Yao. "Architecture for Large Microelectronic Supervised Learning Artificial Neural Networks Using a Hybrid Digital-Analog Approach." *Abstracts of the 1st Annual INNS Meeting, (Special Supplement Issue)*, Boston, Sept. 6-10, 1988. Pergamon Press, NY. p. 389.

[45] Murray, Alan F. "Pulse Arithmetic in VLSI Neural Networks." *IEEE MICRO*, December 1989, pp. 64-74.

[46] Beerhold, J.R.; Jansen, M.; and R. Eckmiller. "Pulse-Processing Neural Net Hardware With Selectable Topology and Adaptive Weights and Delays." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 569-574.

[47] Atlas, Les E., and Yoshitake Suzuki. "Digital Systems for Artificial Neural Networks." *IEEE Circuits and Devices Magazine*, November 1989, pp. 20-24.

[48] Hillis, W. Daniel. *The Connection Machine.* Cambridge, MA: The MIT Press, 1985.

[49] Watanabe, Takumi; Sugiyama, Yoshi; Kondo, Toshio; and Yoshihiro Kitamura. "Neural Network Simulations on a Massively Parallel Cellular Array Processor: AAP-2." *IEEE INNS International Joint Conference on Neural Networks*, 1989. Vol. II, pp. 155-161.

[50] Pomerleau, Dean A.; Gusciora, George L.; Touretzky, David S.; and H.T. Kung. "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second." *1988 IEEE International Conference on Neural Networks*, San Diego, July 24-27, 1988. Vol. II, pp. 143-150.

[51] Annaratone, Marco; Arnould, Emmanuel; Gross, Thomas; Kung, H.T.; Lam, Monica; Menzilcioglu, Onat; and Jon A. Webb. "The Warp Computer: Architecture, Implementation, and Performance." *IEEE Transactions on Computers*, Vol. C-36 No. 12 (December 1989): pp. 1523-1538.

[52] Hayes, John P.; Mudge, Trevor; and Quentin F. Stout. "A Microprocessor-based Hypercube Supercomputer." *IEEE MICRO*, October 1986, pp. 6-17.

[53] Hecht-Nielsen, Robert. "Neurocomputing: picking the human brain." *IEEE SPECTRUM*, March 1988, pp. 36-41.

[54] Deiss, S.; Hicks, W.; Kasbo, R.; Morse, K.; Muenchau, E.; and G. Works. "The SAIC Delta Neurocomputer Architecture." *Abstracts of the 1st Annual INNS Meeting, (Special Supplement Issue)*, Boston, Sept. 6-10, 1988. Pergamon Press, NY. p. 543.

[55] Works, George A. "The Creation of Delta: A New Concept in ANS Processing," *1988 IEEE International Conference on Neural Networks*, San Diego, July 24-27, 1988. Vol. II, pp. 159-164.

[56] Allman, William F. *Apprentices of Wonder: Inside the Neural Network Revolution.* New York: Bantam Books, 1989, p. 112.

[57] Garth, Simon C.J. "A Chipset for High Speed Simulation of Neural Network Systems." *IEEE First International Conference on Neural Networks*, San Diego, June 21-24, 1987. Vol. III, pp. 443-452.

[58] Kato, Hideki; Yoshizawa, Hideki; Iciki, Hiroki; and Kazuo Asakawa. "A Parallel Neurocomputer Architecture towards Billion Connection Updates Per Second." Vol. II Applications Track. *1990 International Joint Conference on Neural Networks*, Washington, DC, January 15-19, 1990. pp. 47-50.

[59] Pacheco, M.; Bavan, S.; Lee, M.; and P. Treleaven. "A Simple VLSI Architecture for Neurocomputing." *Abstracts of the 1st Annual INNS Meeting, (Special Supplement Issue)*, Boston, Sept. 6-10, 1988. Pergamon Press, NY. p. 398.

[60] Kraft, Timothy T., and Stephen A. Frostrom. "Concurrent ANS Architectures using Communicating Processes," Vol. II Applications Track. *1990 International Joint Conference on Neural Networks*, Washington, DC, January 15-19, 1990. p. 51-54.

[61] Hammerstrom, Dan. "A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 537-544.

[62] Duranton, M., and N. Mauduit. "A General Purpose Digital Architecture for Neural Network Simulations." *First IEE International Conference on Artificial Neural Networks*, London, England, October 16-18, 1989. pp. 62-66.

[63] Hirai, Yuzo; Kamada, Katsuhiro; Yamada, Minoru; and Mitsuo Ooyama. "A Digital Neuro-chip with Unlimited Connectability for Large Scale Neural Networks." *IEEE INNS International Joint Conference on Neural Networks*, 1989. Vol. II, pp. 163-169.

[64] Van den Bout, David E., and Thomas K. Miller III. "TInMANN: The Integer Markovian Artificial Neural Network." *IEEE INNS International Joint Conference on Neural Networks*, 1989. Vol. II, pp. 205-211.

[65] Symon, Jim; Rajgopal, Suresh; Miller, Thomas K. III; and David E. Van den Bout. "STONN: A Neural Network IC Using Stochastic Techniques." *Abstracts of the 1st Annual INNS Meeting, (Special Supplement Issue)*, Boston, Sept. 6-10, 1988. Pergamon Press, NY. p. 412.

[66] Wike, William; Van den Bout, David; and Thomas Miller III. "The VLSI Implementation of STONN." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 593-598.

[67] Van den Bout, David E., and Thomas K. Miller, III. "A Digital Architecture Employing Stochasticism for the Simulation of Hopfield Neural Nets." *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 5 (May 1989): pp. 732-738.

[68] Van den Bout, David E., and T.K. Miller. "A Stochastic Architecture for Neural Nets." *1988 IEEE International Conference on Neural Networks*, San Diego, July 24-27, 1988. Vol. I, pp. 481-488.

[69] Tomlinson, Max Stanford Jr.; Walker, Dennis J.; and Massimo A. Sivilotti. "A Digital Neural Network Architecture for VLSI." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 545-550.

[70] Habib, Mahmoud K., and H. Akel. "A Digital Neuron-Type Processor and Its VLSI Design." *IEEE Transactions on Circuits and Systems*, Vol. 36, No. 5 (May 1989): pp. 739-746.

[71] Yestrebsky, Joe; Basehorn, Paul; and Jerry Reed. *Neural Bit-Slice Computing Element.* Lake Mary, FL: Micro Devices, n.d., pp. 1-6. Internal Report, TP 102600.

[72] Cleary, John G. "A Simple VLSI Connectionist Architecture." *IEEE First International Conference on Neural Networks*, San Diego, June 21-24, 1987. Vol. III, pp. 419-426.

[73] Marchesi, M.; Orlandi, G.; Piazza, F.; Pollonara, L.; and A. Uncini. "Multi-Layer Perceptrons with Discrete Weights." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 623-630.

[74] Vidal, Jacques J.; Pemberton, Joseph C.; and James M. Goodwin. "Implementing Neural Nets with Programmable Logic." *IEEE First International Conference on Neural Networks*, San Diego, June 21-24, 1987. Vol. III, pp. 539-545.

[75] Kung, S.Y., and J.N. Hwang. "Ring Systolic Designs for Artificial Neural Nets." *Abstracts of the 1st Annual INNS Meeting, (Special Supplement Issue)*, Boston, Sept. 6-10, 1988. Pergamon Press, NY. p. 390.

[76] Kwan, Hon Keung, and Pang Chung Tsang. "Systolic Implementation of Multilayer Feed-forward Neural Network with Back-propagation Learning Scheme." Vol. II Applications Track. *1990 International Joint Conference on Neural Networks*, Washington, DC, January 15-19, 1990. pp. 155-158.

[77] Diamond, J.; McLeod, R.; and W. Pedrycz. "A Fuzzy Cognitive System: Examination of a Referential Neural Architecture." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 617-622.

[78] Yasunaga, Moritoshi; Masuda, Noboru; Asai, Mitsuo; Yamada, Minoru; Masaki, Akira; and Yuzo Hirai. "A Wafer Scale Integration Neural Network Utilizing Completely Digital Circuits." *IEEE INNS International Joint Conference on Neural Networks*, 1989. Vol. II, pp. 213-217.

[79] Yasunaga, Moritoshi; Masuda, Noboru; Yagyu, Masayoshi; Asai, Mitsuo; Yamada, Minoru; and Akira Masaki. "Design, Fabrication and Evaluation of a 5-inch Wafer Scale Neural Network LSI Composed of 576 Digital Neurons." *1990 International Joint Conference on Neural Networks*, San Diego, June 18-21, 1990. Vol. II, pp. 527-535.

[80] Various authors. *IEEE INNS International Joint Conference on Neural Networks*, 1989. Vol. II, pp. 575-635.

[81] Hopfield, J.J. "Neural Networks and Physical Systems with Emergent Collective Computational Abilities." *Proceedings of the National Academy of Sciences*, USA 79, 1982, pp. 2554-2558.

[82] Denyer, Peter B. *VLSI Signal Processing: A Bit-Serial Approach.* Reading, MA: Addison-Wesley Publishing Company, 1985.

[83] Ma, Gin-Kou, and Fred Taylor. "Multiplier Policies For Digital Signal Processing." *IEEE ASSP Magazine*, January 1990. pp. 6-20.

[84] Ullmann, Jeffrey D. *Computational Aspects of VLSI.* Rockville, MD: Computer Science Press, 1984.

[85] Rumelhart, David E.; Hinton, Geoffrey E.; and Ronald J. Williams. "Learning Representations By Back-propagating Errors." *Nature 323*, (1986): pp. 533-536.

[86] McClelland, James L., and David E. Rumelhart. *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises.* Cambridge, MA: The MIT Press, 1988.