Implementation Limits For Artificial Neural Networks

Thomas Edward Baker
B.S., Oregon State University, 1980

A thesis submitted to the faculty of
the Oregon Graduate Institute
of Science and Technology
in partial fulfillment of
the requirements for the degree
Master of Science
in
Computer Science and Engineering

February, 1990

The thesis "Implementation Limits For Artificial Neural Networks" by Thomas E. Baker has been examined and approved by the following Examination Committee:

Dan Hammerstrom, Thesis Advisor
Associate Professor

Ron Cole
Associate Professor

Todd Leen
Assistant Professor

I would like to thank my advisor Dan Hammerstrom, and my wife Tamara Newton-Baker, for making this thesis possible.

Table of Contents

List of Figures

List of Tables

ABSTRACT

Implementation Limits for Artificial

Neural Networks

Thomas Baker

Oregon Graduate Institute of Science and Technology, 1990

Supervising Professor: Daniel Hammerstrom

Before artificial neural network applications become common there must be inexpensive hardware that will allow large networks to be run in real time. It is uncertain how large networks will do when constrained to implementations on architectures of current technology. Some tradeoffs must be made when the network models are implemented efficiently. Three popular artificial neural network models are analysed. This paper discusses the effects on performance when the models are modified for efficient hardware implementation.

This research supported in part by SRC grant 86-10-097.

CHAPTER 1

Introduction

Recent research has shown that artificial neural networks (ANNs) are a promising solution to many applications that are difficult for conventional algorithms. The parallel distributed processing models are a radical departure from previous artificial intelligence solutions. Instead of processing symbolic data, the ANN algorithms use highly parallel computation to provide problem solutions. Because the ANN models are based on biologically inspired models, they promise to do tasks that are natural for humans but difficult for current computers.

Although the ANN models are biologically inspired, most are not accurate simulations of real neural processes. Simulating the bio-electrical responses of neurons takes a prohibitive amount of CPU time. Compromises must be made for ANN simulators that run on digital computers. Instead of reverse engineering the brain, the ANN models try to achieve similar results with current computational technology.

The ANN research community needs computer hardware that allows faster implementation of ANN models than is currently available. Simulating ANN's on current hardware is slow because of the large number of computations involved. Research is bound by the time limits that current computers place on ANN simulations. Researchers currently cannot simulate more than a few thousand neurons because the execution time of larger networks is too long. Many authorities believe that the models will not be useful until large networks are implemented ($> 10^4$ neurons)[1]. Hardware must be created that will execute ANN models in real time before many useful applications can be developed.

This thesis analyzes whether ANN models can be implemented efficiently with current digital technology. Analog processors are not considered because the precision of memory storage is currently insufficient for the learning algorithms. However, many of the issues that are discussed apply to both analog and digital computers.

There are many tradeoffs that must be made when an algorithm is mapped to a hardware architecture. Some changes that make ANN models run faster on digital hardware negatively affect the behavior of the networks. The tradeoffs should be considered before ANN architectures are designed. This thesis describes what important concessions must be made, and what the performance effects are.

Most ANN models were not designed for efficient hardware implementation. The mathematical models are created by researchers that are not familiar with architecture design problems. Many simulations use double precision calculations that are expensive to implement in a highly parallel architecture. Architectures that use limited precision calculations can be made smaller and faster.

To have fast execution of the ANN models, the algorithms must be mapped to a more efficient architecture. Current technology cannot emulate fully connected communication and high precision computation at the speeds required for real time execution. With fully connected communication, every neuron in one group sends a message to every neuron in another group. The planar nature of current silicon technology limits the number of dedicated connections a parallel processing architecture can have [1]. The use of high precision computation requires relatively large and expensive processors. Simpler processors can be made smaller, so that more processors can be implemented within the same silicon area.

The research that this thesis describes answers the question of how some of the ANN models can be changed to allow hardware implementation. The models are analyzed, and some of the areas that prevent the models from being implemented are discussed. Some possible solutions to the problems that inhibit fast execution are also presented. For each possible solution, a simulation has been made to determine the effects on the performance and effectiveness of the resulting ANN. From the simulation results, decisions can be made whether the solutions are effective or not.

There are many tradeoffs that must be made when deviating from the theoretical basis of most ANN models. The main emphasis of this research is how the algorithm's performance is affected when modifications are made. The models sometimes cease to work effectively, but usually there are changes that will allow hardware implementation. Not all of the modification results are successful. Sometimes there is some degree of improvement by one measure, and a loss of performance by another measure. This thesis should help a computer architect make intelligent decisions about what features must be included in an ANN hardware implementation.

CHAPTER 2

ANN Basics

Before specific models can be analyzed, one should have an understanding of ANN computation. Without an understanding of the basics of ANN problem solving, the analysis of the models will be difficult to follow. Some of the abstract ideas behind the mathematics of the ANN algorithms will be explained in this chapter.

In order to discuss what functions the networks perform, the internal workings of the network will be initially ignored. Instead, the external events will be examined. Think of the network as a function that takes an input, and produces an output. The set of possible input values is called the *domain,* and the output values are the *range.* If the domain is larger than the range, then there is a gain of information. In other words, the function has categorized a collection of input values to a smaller set of output values. The function that is performed by the network is called a *mapping* from the domain to the range.

The strong points of ANNs are parallelism, generalization, and fault tolerance. The mapping is performed by a collection of many simple processors. The advantage of using many simple processors is that much of the computation can be done simultaneously by the separate processors. In conventional computer systems, the program must have a specific response to every input in the domain. If an unexpected input occurs, the results may be undesirable. ANN networks are able to produce a mapping of inputs that were not explicitly provided by the programmer. The network will produce a 'best match' solution based on the information available. Producing a good result from previously unseen inputs is called *generalization*. The networks are also fault tolerant because they perform well if a few of the processors are lost. Some quality of the mapping is lost, but the loss is small compared to a conventional architecture. Conventional architectures are not fault tolerant because a loss of functionality of one of its components usually results in a complete failure of the device.

2.1. High Dimensional Spaces

In order to describe the mappings that the ANNs do, we must discuss the characteristics of the individual components. Many features of an ANN can be defined as high dimensional spaces. The best way to describe the workings of a high dimensional space is to explain the analogy in a low dimensional space, and then extend the model to higher dimensions.

Consider an equation with two variables x and y, where each variable is bounded by finite limits. If a rectangle is drawn on a sheet of graph paper with the sides of the rectangle drawn on the upper and lower limits of the variables, then the area inside the rectangle is the *space* of the variables. The space of a set of variables is the region that can be defined by all the possible values of the variables within the specified bounds. We say that the space of these particular variables has two *dimensions*. The number of different variables defines the dimension of the space. If we add another variable z that indicates the height above the paper, then the space has three dimensions. We can imagine a box that surrounds the bounds of the three dimensional space.

As more variables are put into an equation, the higher the dimensionality will be. A space with a large number of dimensions is often called a *hyperspace*. The prefix *hyper* refers to mathematical objects that have an arbitrary dimension, often the dimensions are greater than three. A system with n variables is called an *n-space*. It is difficult to visualize spaces with more than three dimensions, but many of the spaces that we will examine have many more dimensions than three. The best way to understand how an equation affects a hyperspace is to reduce the dimensionality of the space so that the effects can be visualized. Even if we can not visualize a hyperspace, we can understand the effects of an equation by extrapolating an understanding of a low dimensional space. In the explanation of the ANN models that follow, the algorithms are described in a small dimensional space so that the effects of the equations can be better understood.

In this thesis two types of dimensions are used in a space, *binary* and *continuous*. A binary dimension (or variable) only has two possible values ($x \in \{\alpha,\beta\}$). The values of a binary variable are usually zero and one ($x \in \{0,1\}$), but some binary variables have different values (ex. $x \in \{-1,+1\}$). A continuous dimension (or variable) has a real value ($x \in R$), and theoretically there are an infinite number of values that the dimension can have. Of course, the computer representation of a continuous space must be quantified to a finite number of values.

N Dimensional Objects

The shape of a space is defined by the values that describe the space. Many mathematical objects that are defined in 3-space (our Cartesian world) can be extrapolated for a hyperspace. For example, a *hypercube* has all of the properties of a normal cube except that it can have any number of dimensions. All of the edges of a hypercube have the same length, and all of the dimensions are orthogonal to each other. Consider a three dimensional binary space, and each dimension can have the values of zero and one. The shape of the space is a 3-cube with each edge having a length of one. The space is only on the corners of the cube, because the dimensions cannot have values between zero and one. Adding another dimension to the space creates a 4-cube. It is difficult to visualize a 4-cube, but one can imagine its properties by thinking in terms of a 3-cube.

A *hypersphere* is a space with an arbitrary number of dimensions that has the same properties as a sphere. All of the points on the surface of a hypersphere are the same distance from the origin. The surface of a hypersphere is called a *hypersurface*. A hypersurface usually has one less dimension than the space that it is in. For example, the surface of a sphere has two dimensions. Consider the surface of a globe, the dimensions are north/south and east/west. One can not go in or out of the surface of the globe, even though the surface is curved in 3-space. An important type of hypersurface is a *hyperplane*. A hyperplane is a hypersurface that is flat or linear. The sides of a hypercube are hyperplanes.

Vectors

A point in a space is defined by a *vector*. A vector that defines a point in an n-space has n components, one component for each dimension in the space ($X = \{x_0, x_1, \ldots, x_{n-1}\}$). The values in the vector describe the position along each coordinate of the space. In the above example of the two dimensional space, a two valued vector would define a point on the graph paper within the rectangle. One value of the vector would indicate the distance in the x dimension, and the other value would indicate the distance in the y dimension. Vectors can be either *absolute* or *relative*. An absolute vector refers to the position with respect to the origin of the space. A relative vector uses another position as a starting point. Relative vectors often are used when describing how much to change the current position in the space.

The ANN algorithms map an *input vector* to an *output vector*. In other words, the mapping done by an ANN is from a point in the *input space* to a point in the *output space*. For example, consider trying to map the problem:

$$f(x_0, x_1) = \begin{cases} 0 \text{ if } x_0 * x_1 < 10 \\ 1 \text{ if } x_0 * x_1 \geq 10 \end{cases} \tag{2.1}$$

Where $0 \leq x_0, x_1 \leq 10$. The input space has two dimensions and is continuous, and the output space has one dimension and is binary. We can visualize this mapping if we have a two dimensional graph with the axes of the graph being the input vector and a line drawn on the graph being the dividing line between the binary values of the output.

Many ANN algorithms do not initially perform the desired function. Some problems are too complex to analytically define the mapping. Quite often there is an algorithm that allows the network to learn the correct mapping. In order to learn the proper mapping there must be a measure of correctness, or an indication of the amount of error in the output vector. The difference between the desired mapping and the actual mapping is called the *error space*. The error space has the same dimension as the output space. An objective of the ANN learning algorithms is to find the lowest point in the error space. If the algorithm can find the origin of the error space, then a perfect mapping has been found.

2.2. ANN Computation

Now that we have an idea what ANNs do, we can look at how they do it. Most ANN algorithms are variations of one simple computational model. The calculations of the system are distributed across a network of many independent artificial neurons. The artificial neurons, or connection nodes (CNs), are connected to each other with variable connection strengths. The mapping of the network is contained in the connections between the nodes, not in the control flow of the processors. The discussion of general ANN models will begin with a description of the computation of the individual CNs, followed by a description of how the interconnections between the nodes affect the network, and how groups of nodes work together. Finally, different ways in which the connection strengths can be modified will be explained.

A Single Node

The calculation of the output of an ANN processor is known as the *activation function*. A good way to explain the functionality of the processors is to describe it in the terms of some adaptive signal processing theories. The ANN models have used much of the signal processing computation as a theoretical foundation. The signal processing analogy for the ANN node is the *linear combiner*[2]. The linear combiner is a simple sum of products processor. The processor has an input vector (X), and a weight vector (W). The weights define the connection strength between the inputs and the processing node. The calculation of the output of a linear combiner with k inputs is

$$o = \sum_{i=0}^{k-1} x_i w_i . \tag{2.2}$$

Where x_i is the value of input i, and w_i is the connection strength between input i and the linear combiner. As with the linear combiner, the sum of the inputs times the weights is the basis for the computation of most ANN processors.

The linear combiner is used to define a hyperplane that divides the input space into two regions. The hyperplane is the surface where the output of the linear combiner equals zero. The weight of each connection controls the slope of the hyperplane along the dimension of the corresponding input. If the weight is large then the input has a big affect on the output of the node, and if the weight is small then the input is less significant. If the weight is negative, then the input will inhibit the output of the node.

The function that the linear combiner executes is a *linear function.* A linear function can only create a hyperplane. Sometimes it is desirable to have a node execute a *non-linear function.* A non-linear function is more complex than the linear function and can define a hypersurface that curves through the input space.

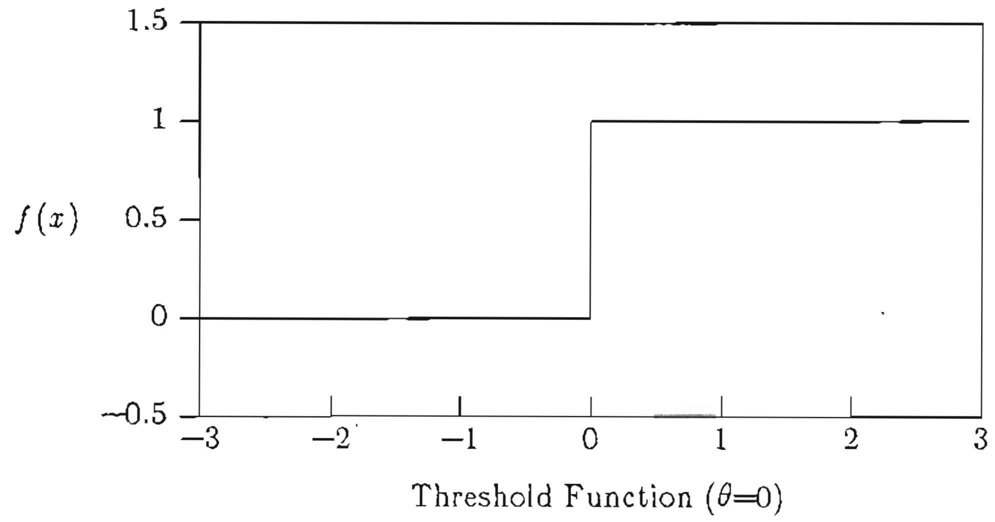

Figure 2.1 - Figure 2.1 A non-linear threshold function.

The most common way to make an ANN node activation function non-linear is to add a function to the output of the linear combiner. One simple way is to add a *threshold* function. A threshold function is often used to make the output of a node a binary value. If the result of the summation is above a given value (or threshold) then the output has a value of one. If the result is lower than the threshold, then the output has a value of zero. We will call this type of node a *threshold element.* One disadvantage of the threshold element is that the error space is not continuous. The output of the node is either right or wrong, there is no way of determining the amount of error because a sum that is close to the threshold has the same output as one that is far away from the threshold.

In order to have a continuous error space, a continuous non-linear function is used. One common continuous non-linear function is

$$o = \frac{1}{1+e^{-C}} \tag{2.3}$$

where C is the result of the linear combiner. Figure 2.2 shows the shape of the non-linear sigmoid function in equation 2.3. In addition to being non-linear, equation 2.3 has several other useful properties. The upper limit of o as C approaches infinity is 1 and the lower limit is 0, although the output

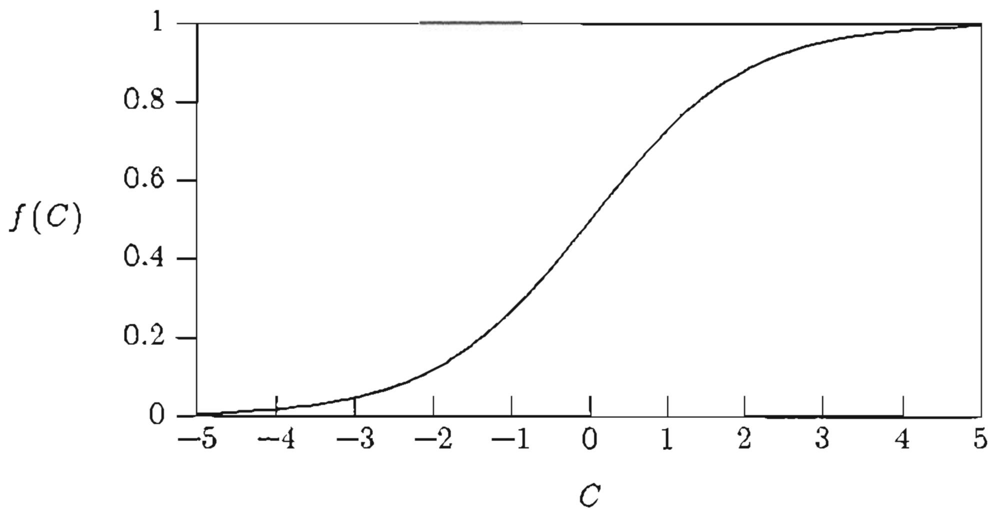

Figure 2.2 - A non-linear squishing function.

never reaches the limits. Having limits on the output of a node is desirable because then there are bounds on the space, and bounds on the input space of the nodes that are connected to it. If the inputs of a node are not bounded, then a very large value for an input can saturate the summation and cancel out the effect of the other inputs. Another property of equation 2.3 is that it is almost linear when C is close to 0, which provides an almost linear error space for nodes that have low values for C. So when C is low then a small variation in the linear combiner output has a large effect on the output, and when C is large (either positive or negative) then a small variation of the linear combiner output has little effect on the output of the node. Equation 2.3 is sometimes called a *squishing* or *sigmoid* function because it makes an infinite input fit into a bounded output space.

There are many types of activation functions that are used with ANN models, but most of them are based on the linear combiner. Which function to use depends on the desired properties of the ANN nodes and the network in general. The specific activation function for each ANN algorithm that is examined in this thesis will be discussed later. A general understanding of the computation of the individual nodes is sufficient for understanding how networks perform their mappings.

Layers

ANN models often group the nodes into layers. The simplest network that we will describe has only two layers, an input layer and an output layer. The nodes of the input layer do not do any computation, they present the input vector to the network unchanged. The output nodes of the ANN models examined in this thesis calculate the activation function by connecting each node in the input layer to each node in the output layer. The result is a mapping onto the output space. The complexity of the mapping done by one layer of a network is limited, because each output activation function can only define one hypersurface in the input space. A single hypersurface may not be able to map disconnected regions into the same classification.

For complex mappings, more than two layers are required. Additional layers are called *hidden layers* because they are not accessible to the external world. In a three layer network, the hidden layer is connected to the input layer and the output layer is connected to the hidden layer. The purpose of the hidden layer is to do a partial mapping of the input vector. The hidden layer reduces the complexity of the input vector by dividing the input space into regions. Each node in the hidden layer defines a hypersurface that splits the input space into two regions. The combination of all the hypersurfaces defined by the nodes in a hidden layer can be used to simplify the input space. Theoretically, most mappings can be done by a network with only one hidden layer[3]. However, in practice it is often useful to use more than one hidden layer. Each hidden layer reduces the complexity of its input space until the space is simple enough to map with the final output layer.

ANN Learning

The most difficult aspect of making an ANN application is determining what the weights should be, because the weights contain the mapping of the network. Each ANN model has a different type of learning algorithm, all of which try to reduce the error space. The difficulty of the learning functions are that they must solve the credit assignment problem, which has been a problem for as long as researchers have tried to make computers learn. In terms of ANNs, the credit assignment problem concerns determining which weights contribute to successful solutions and which weights inhibit the network from performing correctly. A learning algorithm modifies the weights, and may not be modifying the right ones. Determining the credit for hidden nodes is particularly difficult, because there is usually no explicit information about what the intermediate activation values should be. The specific learning algorithms will be discussed later. The discussion in this section will cover what is learned and the different categories of learning algorithms.

The ANN algorithms can only learn what is presented to the networks. A network simulation is presented a set of training data. The learning algorithm then modifies the weights of the network to best map the training data. Different sets of training data produce different results. If the training data are chosen improperly, then the network may not learn the desired mapping. For this reason it is important that the training data be chosen carefully. The mapping that is learned will usually be a representation of the input distribution of the training set, and may not extrapolate well to new data that are not in the training set.

Some ANN learning algorithms require that a training vector is presented along with the input vector. If the training vector is the desired output, then the weight modification function is a *supervised learning* algorithm. With supervised learning there is a well defined error space. One of the problems with supervised learning is that the correct mapping for a problem might not be known. Other learning algorithms just have a global value for correctness, which is called *reinforced learning.* While reinforced learning has a simpler error space (right and wrong), the learning algorithms are more complex. We do not examine any reinforced learning models in this thesis.

The last type of ANN learning does not have any training input at all, these algorithms are called *unsupervised.* Each layer of a network that implements unsupervised learning attempts to distinguish features of its input space. The learning algorithm uses an internally generated error measure to modify the weights. A network that is not supervised must have an external algorithm if a specific input-output mapping is desired. Without a training signal, the network has no concept of what the desired output space is. Unsupervised and supervised learning algorithms are sometimes used together[4]. The unsupervised layer can be used to reduce the complexity of the input space, and the supervised layers use the output of the unsupervised layer to make the final mapping to the output space.

CHAPTER 3

An Artificial Neural Network Architecture

The goal of the Cognitive Architecture Project (CAP) at the Oregon Graduate Institute is to create wafer scale neurocomputers[5]. There are many architectural issues that must be resolved before our goal can be realized. Some of the research by other members of the CAP group involve exploring the problems of building a massively parallel system. The research in this thesis is concerned with some of the performance tradeoffs that occur when ANN algorithms are approximated with silicon implementations. This chapter will discuss the relationship between architecture issues and ANN algorithm issues.

It is not the purpose of this thesis to design a computer architecture that will run ANN models. However, there must be a general target architecture to simulate. The model architecture must be general enough to emulate different types of ANN algorithms. The architectural design issues must be examined carefully or the simulation results will have little meaning.

The main focus of this chapter is to discuss which computer architectures are best suited for executing ANN models. Several factors suggest that some type of parallel architecture is a good solution. Many of the ANN models simulate distributed nodes, with each node executing independently of the other nodes. The ANN nodes can operate in parallel. Although the individual activation functions are usually simple, there is a large number of computations required to simulate an entire network because of the number of nodes in a network. Because of the need for a large number of computations to execute the algorithms, and the ability to execute in parallel, a parallel architecture is a natural solution for an ANN system.

CNs vs. PNs

The distinction must be made between Connection Nodes (CNs) and Processor Nodes (PNs). A CN refers to an artificial neuron as discussed in the previous chapter. The information used to define the mapping performed by an ANN is contained in the connections between the CNs. A PN is a single processor in a computer system. A PN is normally one of many cooperating processors. The relationship between CNs and PNs depends on the ANN network structure and the computer architecture. At one extreme, all CNs can be processed by a single PN, which is the way most ANN simulators are implemented. If there are more CNs in the ANN than there are available PNs, then the PNs must multiplex the CNs. At the other extreme, multiple PNs can process the output of a single CN[6]. The latter extreme is used by systems that have very simple PNs. Theoretically, complete parallelism can be obtained by using one PN per connection. However, implementation issues restrict the amount of parallelism that is practical. Real neurons are able to have three dimensional connectivity, while current silicon implementations are limited to two dimensions.

When considering the individual PNs that will make up the parallel architecture, a major concern is the size of each PN. Any PN architecture requires a finite amount of space. There is a tradeoff that must be made between the size of the individual PNs and the number of PNs that can fit into the system. For example, floating point adders and multipliers are generally large compared to equivalent integer units, but a particular application may require the dynamic range of values that only a floating point representation can provide. So the architect must use care when deciding what capabilities should be provided by the PNs.

Many ANN researchers find analog computation appealing[7]. Analog arithmetic units can be made smaller and faster than digital hardware, but there are also disadvantages to using analog computation. Current analog technology is limited by the precision that can be accurately represented. Analog data storage is also unreliable for long periods of time due to the thermal properties of silicon. Analog weights tend to decay when used in the normal temperature ranges. For these reasons, digital integer calculations are used to simulate the ANN algorithms. However, the simulation results should apply to analog calculations if the technology permits the precision specified in the results.

The PN Model

The PN architecture that is assumed in this thesis is a simple but fast digital processor. Each PN will have an integer adder and a multiplier that can execute in parallel. With parallel addition and multiplication the PNs can sum the result of each connection every clock cycle. The execution of multiply and accumulate (MAC) operations every clock cycle is common for current digital signal processors. The MAC operation is the calculation performed by the linear combiner (equation 2.2). The PN architecture can calculate one connection (input times weight) per clock cycle. It is also assumed that the simulated processor has the capability to perform other operations. When estimating the execution of the hardware, every operation can occur in one clock cycle. For the simulations described in this thesis, there is one PN for every CN, so that all the CNs in a particular layer can be calculated simultaneously.

The proposed PN architecture is very similar to current Digital Signal Processor (DSP) technology. One of the primary uses for modern DSP chips are to execute parallel MAC operations. There is also enough functionality on a DSP processor to perform most general purpose operations. It is assumed that the proposed PN architecture has the functionality to perform whatever simple digital operations are required by the ANN algorithm. However, if an ANN algorithm requires an operation that is expensive to implement in silicon, an attempt will be made to modify the algorithm so that the PN architecture can remain simple. It is also assumed that the arithmetic units have enough precision to perform the algorithms to be discussed. Attempts will be made to determine how much precision is required by the arithmetic units of the PNs. Each PN will also have internal registers for storing intermediate calculations, and local memory for weight storage.

The correct selection of which parallel architecture to use is a difficult problem. The major drawback to silicon implementations of ANN models is that a processing node must send messages to many other nodes. Mapping the highly connected networks to a planer representation is not practical with one (or more) electrical circuit per connection [8]. Other members of the CAP group have produced workable solutions to the connectivity problem by multiplexing the information passed between processors [9]. This thesis is not concerned with the method of transferring the connection information, rather it will be assumed that a suitable method is available. There are many computer network topologies that can provide inter-PN communication.

CHAPTER 4

The CAPsim Simulation Library

Until special hardware is created for ANNs, the algorithms must be simulated on conventional architectures. The target architectures and which ANN algorithms are implemented are important factors when selecting or designing a simulator. There are many types of ANN simulators that have been designed by various researchers. Each simulator has advantages and disadvantages. Some simulators were made to run specific ANN algorithms, while other simulators were made to run on a particular computer architecture. There are also simulators that are designed to run any ANN algorithm on a wide variety of computers. The hardware or algorithm specific simulators are usually faster because they are optimized for the particular implementation, however optimization limits the portability of the simulators. General purpose simulators are more portable, but they are usually less efficient.

The research of this thesis requires a simulator that has flexibility and speed. The simulator must be able to execute several different ANN algorithms. However, the algorithms have similar characteristics. The algorithms analyzed in this thesis have the nodes grouped in layers, and all of the networks have at least two layers. Each node also receives input from every node in the previous layer, and there are no connections between nodes in the same layer. The above restrictions to ANN algorithms allow some optimization of the simulator, while these optimizations may restrict the simulator from efficiently executing algorithms that do not have the above connectivity requirements.

The simulator must also be flexible in the type of computation that is simulated. This thesis compares the affects of limited precision integer computation with floating point computation. The simulator must be able to run the same applications using both types of computation, and it should not be too difficult to switch between the two data types. The fact that the integer precision must be adjustable adds complication. The integers must represent decimal values in a fixed point representation. Every calculation that is made must adjust the results so that the binary point is in the proper location.

The simulator must be as fast as possible considering the restrictions discussed above. Since the nodes are grouped in layers with full connectivity between layers, the activation values and weights can be stored as linear arrays. With the data stored linearly, the calculation of the activation values can be vector operations. Although the simulator does not run on a vector processor, the code can be written to run faster than if the layers were not fully connected.

CAPsim

At the time that the research for this thesis began there were no available simulators that met the above requirements, so the CAPsim library was created. CAPsim is an ANN simulator library. CAPsim is not a single network simulator, nor is it a program that allows one to dynamically create networks and topologies. CAPsim is a library of simulator modules that can be used to create an ANN application. The CAPsim library consists of many useful building blocks for neural network simulators. The main purpose of CAPsim is to study the effects of hardware implementation on neural network algorithms. It can also be used to create and simulate ANN applications.

The two strongest considerations in the design of CAPsim are code modularity and execution speed. The simulators that are implemented with the CAPsim library need to have many simple modules, so that they can be easily constructed and interfaced cleanly and flexibly. The library should help the programmer build networks with only a few definition statements. However, the modularity of the library should not slow down the execution of the simulations. Most of the execution time for the simulators are spent in a few small computation loops. Although most of the CAPsim library is programmed for readability, the computation loops are programmed for speed.

The C++ programming language was chosen as an implementation language because of the combination of object oriented language features and efficient target code. The C++ language is a superset of the C programming language. The features of C++ combine the fast execution of the C language with object oriented paradigms.

One basic data object of the C++ language is the *class*. A class is a data structure that has state (data) and a set of operations to be performed (member functions) on the state. The class construct allows programs to be built and tested modularly. Normally, the only way that a class's state can be accessed is through its functions. The member functions of a class can be programmed and tested independently from the rest of the program. Classes are separate software modules that can be used to quickly construct a simulation.

The object feature of inheritance .makes the task of programming modular code much easier. Inheritance is a way of defining hierarchies of classes. New classes are derived from an existing class, and the new class will inherit the same state variables and member functions as the class from which it was derived. The class that originally defines the code is referred to as the *superclass,* and the class that inherits the characteristics of the superclass is known as the *subclass.* Inheritance lets a subclass reuse code that has already been written for its superclass. The subclass can define new state variables and member functions, or redefine the member functions derived from the superclass. For example, a class can be written that will define the interface protocol between two software modules. The communication functions can be written once, so that the interface functions do not have to be rewritten for each new module that is programmed. All classes that inherit the member functions can use the inherited code to communicate.

A feature of C++ that helps produce fast programs are *inline functions.* During the compilation of a program, the code that would normally be executed in a procedure is inserted directly into the code. With inline code, the time consuming task switch (and parameter passing) of a function call is no longer present. Most of the execution time for the simulators occurs in calculations in a few loops. Inline functions are used to keep function calls within these loops to a minimum. The use of inline functions helps make the code within the calculation loops readable, modular and fast.

The execution time for many of the simulations is from several hours to several days, so the user interface has been designed for both interactive execution and batch processing. The emphasis for the user interface is to provide functions for setting key simulation variables, tracing intermediate values, and running the simulator. The simulation variables define the algorithm modifications that are to be simulated without the need to recompile the code. While the simulation executes, trace values are printed periodically to show the performance of the ANN algorithm. The CAPsim library makes custom menus and user input functions easy to program so that the user interface can be tailored to the ANN application.

The class *vector* is the superclass of all ANN objects in the CAPsim library. A vector represents one layer in an ANN network. Vector defines the interface to the values of the nodes in each layer of the network. The vector class is really only an interface specification. The member functions specify how the data which is defined in subclasses of vector are accessed. The vector class interface is designed as a one-dimensional array that contains the activation values of the nodes in the layer. The member functions in the vector class define the operations that can be performed on the array. All ANN application classes inherit the vector class member functions. All values that are sent between layers of an ANN application use the member functions that are defined in the vector class. Each application class reads the values of the layers that it is connected to by using the vector member functions. Although vector is defined as a one dimension layer, higher dimension layers can easily be implemented.

Although the vector class defines the interface between the ANN modules of the simulator, the actual data structures are not defined in vector. The activation values of the layers are defined in the subclasses floatVector and integerVector. These classes contain the arrays of values for the vector. The class floatVector defines an array of type float. FloatVector is used for most general ANN applications. The class integerVector defines an array of type int, and is used for simulating a hardware architecture that uses integer arithmetic. The values contained in integerVector are integer representations of fixed point values. When an ANN algorithm class is created, it is defined as a subclass of floatVector or integerVector.

CHAPTER 5

Back Propagation

One of the most popular ANN models is the back propagation learning algorithm. Back propagation was conceived by several different people independently [10] [11] [12]. The algorithm implements supervised learning for multilayer networks. The network can have an arbitrary number of layers, although most applications use three layers. For an N layer network, the first layer is the input layer and layer N is the output layer. Layers 2 to $N-1$ are the hidden layers, which are not visible to the external environment. A node in layer n $(1 < n \leq N)$ receives inputs from every node in layer $n-1$. The hidden layers develop internal representations of the training data. The power of the back propagation algorithm is that it has a solution to the credit assignment problem discussed in Chapter 2, so that it can learn complex mappings by using more than one layer.

Each layer produces its output by using the non-linear squishing function in equation 2.3. Since back propagation is a supervised learning algorithm, the desired outputs must be known for every input vector that is used in training the network. The results of layer N (the output layer) are compared with the desired outputs and the weights are modified. Weight modification is achieved by using the least mean square (LMS)[2] algorithm. LMS was developed in the 1960's for digital signal processing applications. LMS was originally used with the linear combiner activation function (equation 2.2).

Since the linear combiner is a linear function, the error surface is also linear. The LMS algorithm uses the square of the estimated error surface to find the direction that the weight should be changed. When the error of a linear function is squared, the error surface (with respect to the weights) becomes a hyperparabola. The minimum error (i.e. the best mapping) is at the bottom of the hyperparabola, so the gradient of the squared error surface always indicates the direction of the least error. The weights are iteratively modified by using the gradient of the square of the estimated error.

The key insight of the LMS algorithm is that the error of the linear combiner at any point in input space is a good estimate of the error surface (E). The gradient of the squared error of the linear combiner (equation 2.2) is

$$\nabla E^2 = \frac{\partial E^2}{\partial w_{ij}} = \delta_i x_j \tag{5.1}$$

where δ_i is the error for a node in layer n, w_{ij} is the weight between node i in layer n and node j in layer $n-1$, and x_j is the activation value of node j in layer $n-1$. For the linear combiner, the error is calculated by the equation

$$\delta_i = d_i - o_i \tag{5.2}$$

where d_i is the desired output and o_i is the actual output of a node.

To get the gradient of the non-linear function of equation 2.3 which is used by the back propagation algorithm, the derivative of the squishing function must also be calculated. The derivative of equation 2.3 is:

$$f'(o) = o(1 - o) \tag{5.3}$$

where o is the output value of a node. By using the calculus chain rule, the error for a node in the top layer of a back propagation network is found by

$$\delta_i = o_i(1 - o_i)(d_i - o_i) \tag{5.4}$$

The error for a hidden layer (n) is calculated by multiplying the errors of the nodes in the layer directly above it $(n+1)$ times the value of the weights connecting the two layers.

$$\delta_i = o_i(1-o_i)\Sigma\delta_k w_{ik}. \tag{5.5}$$

So if there is a strong connection between a hidden node and an output node, then the δ of the output node contributes significantly to the error of the hidden node.

The weights are iteratively modified in the direction of the error gradient. The weight update equation is:

$$\Delta w_{ij}(t) = \eta \delta_i x_j + \alpha \Delta w_{ij}(t-1) \tag{5.6}$$

where η is a constant that determines the learning rate. The term $\alpha \Delta w_{ij}(t-1)$ is called the *momentum*, and α is the constant for varying the amount of momentum that the learning equation uses. The momentum allows the weight change to accumulate over several input vectors so the weights that are constantly updated in the same direction can change faster than if just the gradient is used.

A detailed proof of the back propagation algorithm is given in [12], where the authors show that the algorithm will always attempt to decrease the error. However, it is possible for a network to get trapped in *local minima*. Local minima are valleys in the error space that have lower values than neighboring regions, but the local error measure is not the absolute minimum of the error surface. The gradient of the error surface at a local minimum is always in the direction of increasing error, so the network would be drawn towards the bottom of the valley. For high dimension non-linear error spaces that occur in a back propagation network, a local minimum could prevent the network from learning the desired mapping. The momentum term of equation 5.6 helps prevent the network from being trapped in a local minimum by acting like a low pass filter. Imagine a marble rolling down the error surface. The marble would be constantly drawn toward the path of least resistance, but the momentum of the marble would allow it to escape from small valleys.

There has been a lot written about methods for improving the learning speed of the back propagation algorithm. Learning is perceived to be slow because of the iterative nature of the weight updates. For complex problems, a back propagation network may take a large number of input presentations before a solution is reached. Several modifications use complicated equations for adaptively modifying the learning rate[13]. Other modifications attempt to determine the second order derivative to decrease the number of input presentations required for finding the solution[14] [15]. Although the modifications have improved the learning rate, they have also burdened the network with extra computation. If the algorithm modifications decrease the number of input presentations by an order of magnitude (which is seldom the case) but increase the number of calculations by an order of magnitude, then there may be a net loss. The actual real time execution may be the same, but these modifications often require functions that would increase the size of the processor that is executing the algorithm. As discussed previously, our goal is to have many simple processors instead of a few large processors. The back propagation simulations used as a reference for this thesis used the standard algorithm as defined in [12].

Application: Character Recognition

There are two applications that were used for testing the modifications that were made to the back propagation algorithm. One application was a character recognition problem. The training set was a set of characters from eleven different alphabetical fonts taken from the Apple MacIntosh font set (286 training pairs). The characters are shown in figure 5.1. The inputs were a twelve by twelve array forming a pixel representation of a single character. The position of the character was justified to the top left. There were twenty-six outputs, one output for each character in the alphabet. The network was required to match the bit map of a character to the correct output category. There was one hidden layer with thirty nodes. This application is a non-trivial problem that tests the networks' learning capacity and the ability to extract the high order features of the input space. The network was required to memorize the learning set, generalization was not tested. The mapping required of the network was from a binary input space to a binary output space. Supervised networks of this type are normally trained until the maximum error of any single node is within a given ϵ, for all training vectors. The criteria for convergence of this problem set was with ϵ equal to 0.4. If a network converges to an ϵ of 0.4, it will normally converge to a smaller ϵ if the network is allowed to run longer.

Application: Coordinate Transformation

The second application was a coordinate transformation problem. A two axis robot arm was simulated. The inputs to the network were the desired position in Cartesian coordinates, and the outputs were the corresponding axis positions of the robot arm. Figure 5.2 shows a diagram of the simualated arm. This simulation mapped a continuous input space in Cartesian coordinates to a continuous output space in polar coordinates. The representation of each axis was a problem for this simulation, because representing a continuous vector with limited precision values restricts the resolution of the input and output spaces. Even if the nodes had infinite precision, assigning a single node to each axis would create difficulties that were not dependent on the problem. Due to the fact that the output nodes

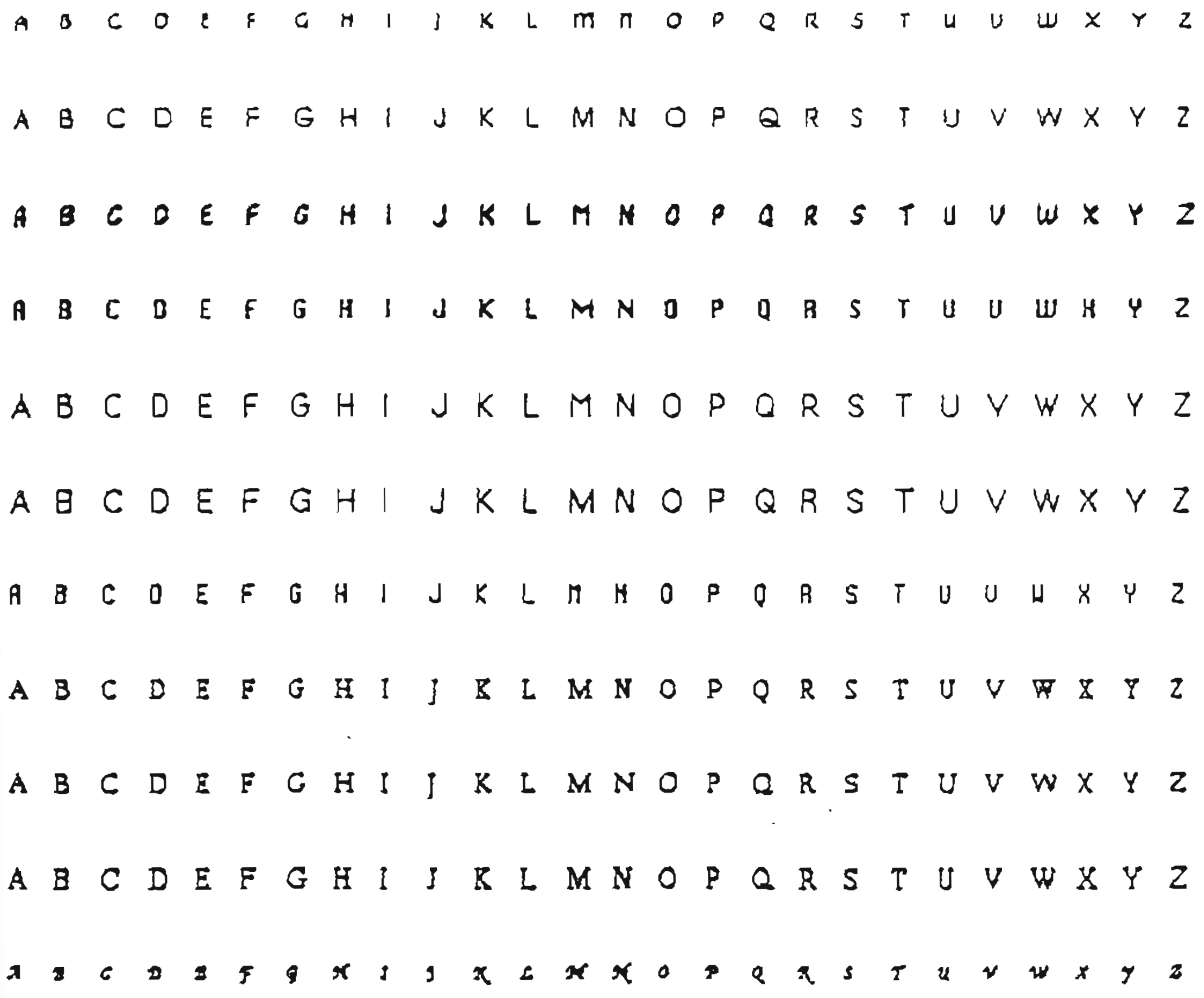

Figure 5.1 - The character recognition training set

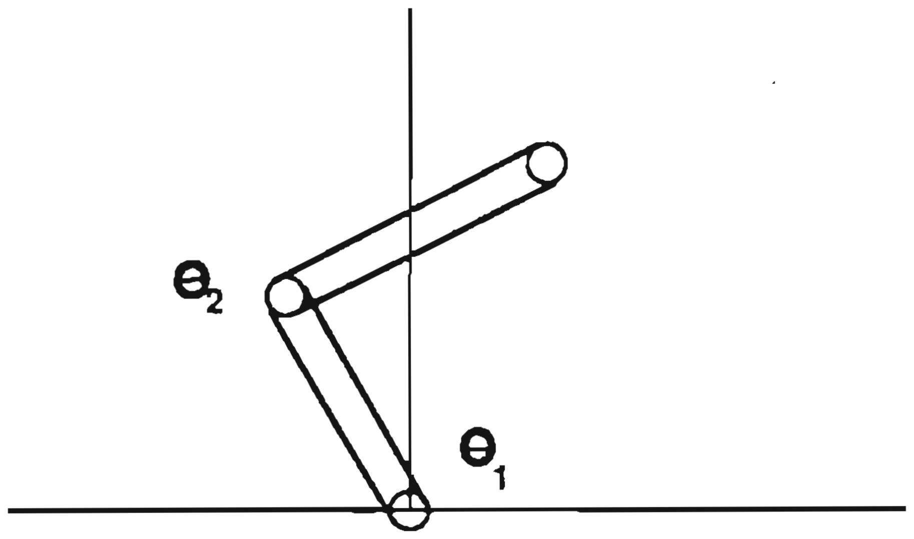

Figure 5.2 - The coordinate transformation robot arm.

are non-linear, equal amounts of change in the activation value would require the network to compensate for the non-linearities inherent in the squishing function. When the output of a CN is near 0.5, a small change in the weights will cause a large change in the output when that input vector is presented again. However, when the output of a CN is near 0 or 1, a small change in the weights will cause almost no change in the output value. For this reason, the input and output space was distributed among several nodes.

The representation that was used for the simulation of the input and output space of the coordinate transform problem had several nodes for each axis. The value of each node corresponded to a limited range of the axis it was assigned to. There were sixteen nodes per axis for the output vector. The output nodes were grouped so that a cluster of four nodes represented the full range of one arm axis (360 °). There were four clusters of nodes, each cluster was rotated so that the range of a node did not correspond exactly to any other node. The input nodes were similarly distributed for the Cartesian coordinates, except that there were twenty nodes per axis. This representation allowed a higher resolution for the axis than any single node could have. and for any particular point on the axis, there were four nodes that had ranges that overlapped.

For the coordinate transformation problem, there were a total of forty input nodes, twenty hidden nodes and thirty-two output nodes. Although the robot arm simulation allowed full rotation of each axis, the training set was restricted to a smaller area of the input space. The area restriction was made so that the network would converge within a reasonable amount of time. Trying to learn the coordinate transformation for the entire reach of the robot arm would take too many training cycles. The problem was constrained so that a solution could be found within several hours. The orientation of the arm was also restricted because there may be two possible robot axis configurations that will put the end of the arm at a point within the input space. If the arm positions were selected randomly, there could be two correct output vectors for each input vector. The restriction of having the elbow axis being always greater that $180\,^{\circ}$ solved the problem of having multiple solutions for each input vector. The training set used a random position for each input presentation. Randomly selecting the training set is a good test of the generalization characteristics of the network, since the network cannot memorize the training set. The convergence criteria for this problem was $\epsilon < 0.15$ for ten consecutive input presentations.

5.1. Limited Precision

Floating point processing units require more silicon area than is feasible for a highly parallel architecture. For a cost effective solution, the PNs should have a limited precision integer multiplier and adder. The PNs could also be an analog processor if the technology provides adequate precision. As discussed in Chapter 3, many researchers believe that analog processors are a good architecture for ANN implementations. The important issue is how much information the back propagation algorithm requires for the activation values and weights.

Fixed point computation can be used for limited precision architectures. The activation value of equation 2.3 can be computed by summing the products of the inputs o_j and the weights w_{ij} with the parallel multiply accumulate (MAC), and a table lookup can be used for the squishing function. By using a table lookup, the binary point of the results can be automatically justified. Because the activation value of each node is always positive and less than one, the binary point will always be to the left of the significant bits. Our simulations have shown that eight bits of precision are enough for the activation values of the outputs.

The weights however, commonly grow larger than one, and can have both positive and negative values. So the binary point for the weights must be placed within the significant bits. There must also be one bit to represent the sign of the value. Three bits to the left of the binary point were used, which limits the weight values to less than eight. The weights of the floating point simulations sometimes grow larger than eight, but weights greater than eight do not seem to be required. Examining the weight values after the successful floating point simulations runs indicates that few weights actually grow larger that eight, but many weights have values between one and eight. In fact, observing the weights of networks that failed to converge show that some of the weights grew to be very large. In the limited precision simulations, if a value overflowed it was set to the maximum (or minimum) value allowed by the precision.

All of the results presented in this thesis are empirical. There are many researchers studying the analytical behavior of ANNs, but most of the published results have been restricted to linear systems. Non-linear systems are more difficult to characterize. Although an attempt was made to find the reason behind the simulation results, a formal analysis is beyond the scope of this thesis.

Limited Precision Results

The simulation results of limited precision calculations for the character recognition problem is shown in Table 5.1. Table 5.2 shows the results of the coordinate transformation problem. In the tables, each value represents the average number of input presentations for the application to converge. Each simulation was run with five different sets of initial weights. A floating point simulation is given for comparison.

Character Recognition
Floating Point vs. Integer

Algorithm Modification	Input Presentations
Floating Point	36,490
16 Bit Weights	21,340

Table 5.1

The limited precision simulation results show that sixteen total bits of precision (one sign bit, three bits to the left of the binary point and twelve bits to the right) does not significantly degrade the performance of the network. The simulations have shown that there is a limit of twelve bits of weight precision required by the algorithm. The twelve bit limit results from the fact that individual weight updates in equation 5.6 are small quantities.

Coordinate Transformation Problem
Floating Point vs. Integer

Algorithm Modification	Input Presentations
Floating Point	26,900
16 Bit Weights	14,960

Table 5.2

The ΔW_i can be thought of as a relative vector in the weight space. With a limited precision weight space, there is a finite number of quantities that each weight value can have. The distance between the quantities of a vector define the precision of the space. If the ΔW_i is smaller than the precision of the space, then the weights will not change. The network will stop learning when the weight updates are smaller than the precision of the weights.

For the back propagation learning algorithm, the small quantities are created because all of the values that are multiplied to get the Δw_{ij} are values less than 1. In fact many of them are small fractions, which creates a very small Δw_{ij}. The algorithm produces weight updates that are smaller than can be represented with eight bits to the right of the binary point ($\Delta w_{ij} < 0.004$). Using fewer than eight bits to the right of the binary point would inhibit learning. The dynamic range of the weights is the cause of the limit for the number of total bits needed for learning. The learning equation uses values that require more than eight bits to the right of the binary point, and the values of weights after the network converges often require bits to the left of the binary point to drive the outputs of the nodes through the full range.

The limited precision simulations required fewer input presentations to learn the training set than the floating point simulations. It is doubtful that limited precision simulations learn faster for all training sets. However, examining the reason that using limited precision simulations improved the performance of these training sets reveals some interesting information about the learning algorithm. One of the problems with the back propagation learning rule is that it tends to overlearn the training set. Overlearning occurs at the end of the training session when the weights are modified even though the network produces an output that is very close to the desired output. The learning algorithm will continue to reinforce the weights. When the desired outputs are 0 and 1, the output of the network will never reach the goals because the sigmoid function (eq. 2.3) never reaches these limits. So the learning algorithm will continue to increase the weights because the outputs can never reach the targets. One solution is to set the desired outputs at 0.1 and 0.9, which will keep the weights from growing infinitely but will not prevent the learning algorithm from reinforcing small errors.

A method for preventing the network from overlearning is to stop modifying the weights when the error for a particular input vector is below a certain value. Then the weights are only changed for input vectors that the network has not learned correctly. Several researchers have used this technique to improve learning time [16]. For the implementation of the limited precision simulations presented here, the behavior of not updating the weights in the event of a small error happens automatically. The δ_is of equation 5.4 were stored as eight bit values. The δ_is were stored with the same precision as the outputs of the hidden layer, because the δ_is are sent on the same communication channel.

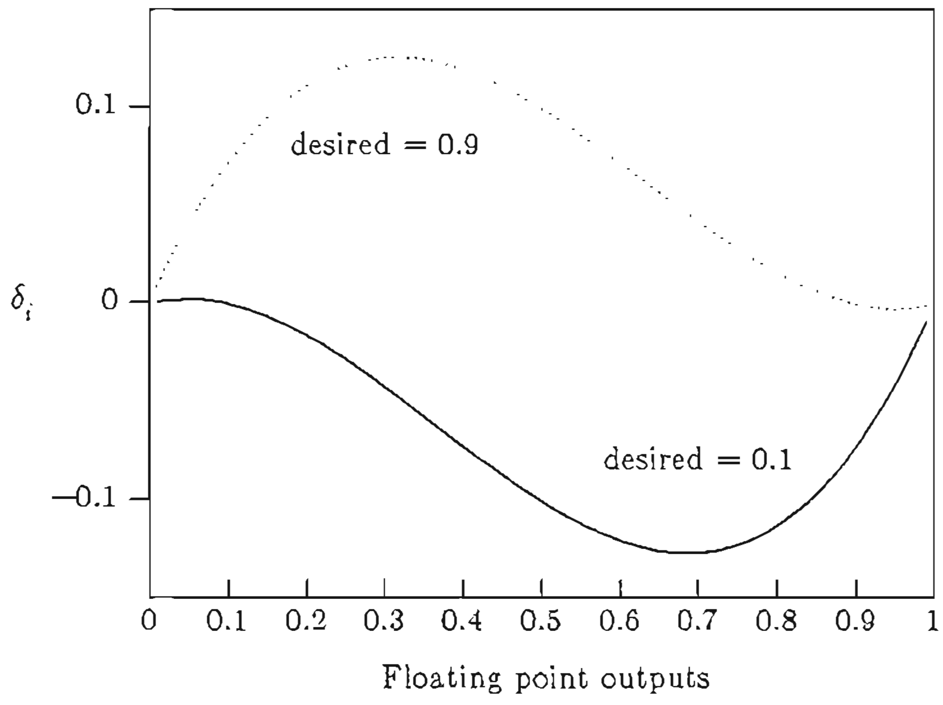

Figure 5.3 - Floating Point Error

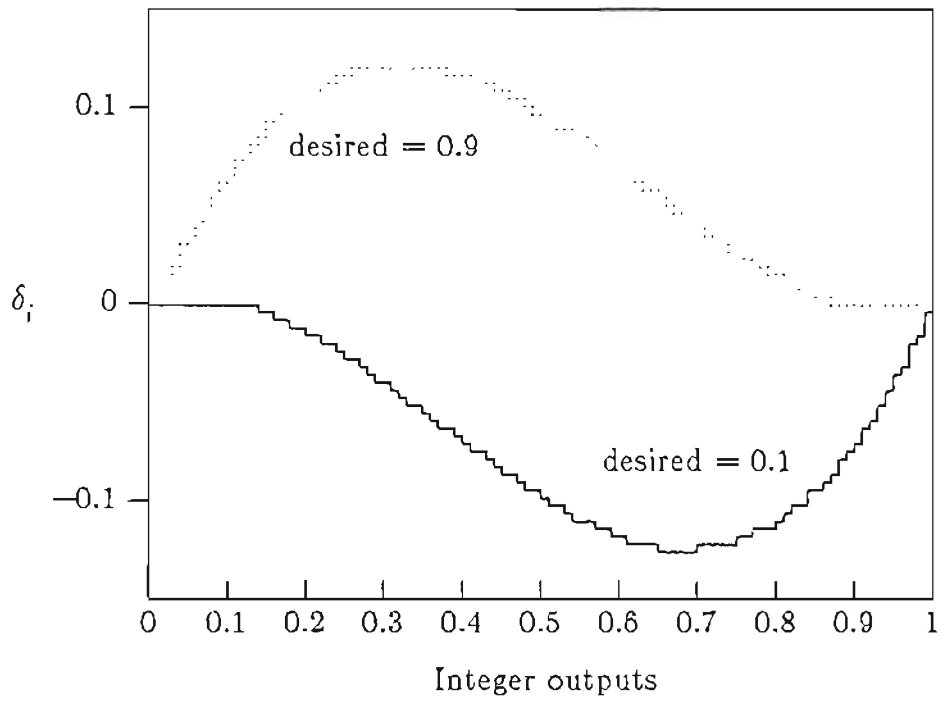

Figure 5.4 - Limited Precision error.

Figure 5.3 shows the floating point value for δ_i over the range of output values with targets at 0.1 and 0.9, and figure 5.4 shows how the limited precision implementation affects the δ_i equation. The limited precision network stops learning when the output value is close to the desired value. To show that limiting the δ_i values when the error is low improves the performance of a network, the floating point simulations were run with the same constraints on the error calculation (as shown in figure 5.5). Tables 5.3 and 5.4 indicate that limiting the δ_i values allows the network to learn with fewer input presentations. Limiting the δ_is caused the floating point implementation of the network to learn the character recognition problem faster than the limited precision implementation. However, the limited precision implementation still learned the coordinate transformation problem faster than the floating point simulation, indicating that there are other side effects of the

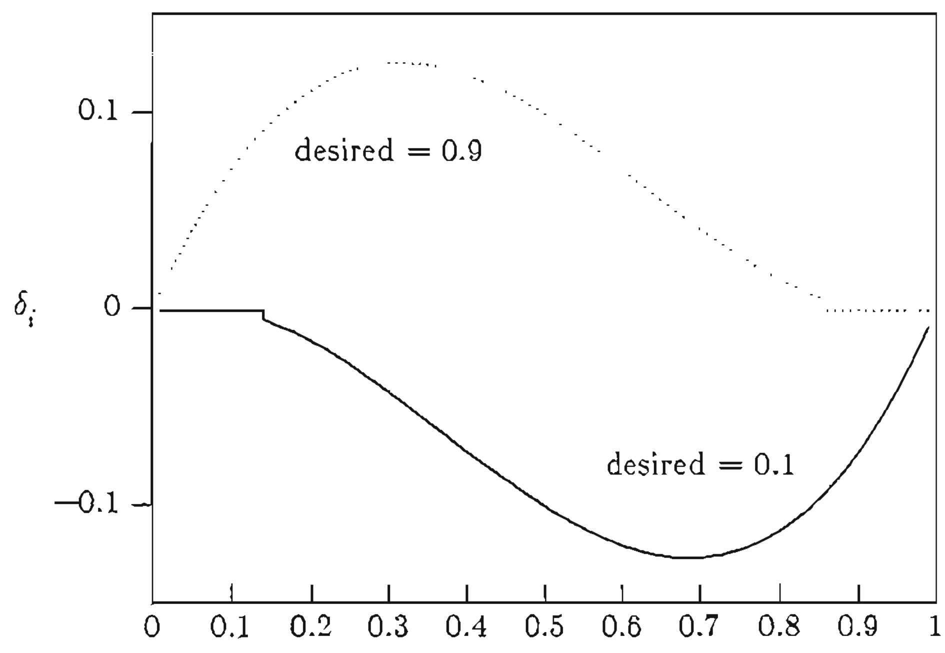

Figure 5.5 - Floating point error with limits.

limited precision implementation that can improve network learning.

One must take care when analyzing the side effects caused by using limited precision calculations. The results presented here show that using limited precision values can increase the performance of the back propagation learning algorithm, but the network will not learn faster for all training sets. If the application requires a small error for convergence, then more precision must be used in the error calculations. Another factor that must be considered is that the free parameters of the simulations were optimized for the limited precision calculations. The learning rate (η), momentum rate (α), target values (d=0.1 or d=0.9), and number of hidden nodes were adjusted for minimum convergence time of the limited precision implementation. This was done because most of the simulations presented in this chapter used limited precision calculations, and an effort was made to reduce the amount of simulation time. The floating point simulations may have converged faster with different parameter values. However, time did not permit exploring the optimal values for each implementation of the algorithm.

Character Recognition
Regular Floating Point vs.
Floating point with δ limits

Algorithm Modification	Input Presentations
Regular Floating Point	36,490
16 Bit Weights	21,340
Floating Point with limits	20,300

Table 5.3

Coordinate Transformation
Regular Floating Point vs.
Floating point with δ limits

Algorithm Modification	Input Presentations
Regular Floating Point	26,900
16 Bit Weights	14,960
Floating Point with limits	24,350

Table 5.4

5.2. Sign/Threshold Propagation

Another problem that would make hardware implementation of the back propagation algorithm difficult is the interprocessor communication required to execute equation 5.5. The information that must be sent between processors in all of the other equations can be broadcast ($O(n)$

communication complexity) from the originating node to all of the other nodes. The hidden nodes, however, must receive a unique weight value from each of the output nodes, which requires point to point messages ($O(n^2)$ communication complexity). The high communication requirements are because the back propagated error accumulation uses the transpose of the weight matrix. Each PN has the fan-in weights in its local memory for fast activation value calculation, and the calculation for the error of a hidden node (equation 5.5) requires the fan-out weights.

Anderson [17] discovered that the network learning performance can increase when the sign of the weight was used in equation 5.5 instead of the actual weight value. Using Anderson's results leads to a useful modification of the basic algorithm. If the sign of the weights for the connections between the hidden nodes and the output nodes are kept in the local memory of the hidden nodes, as well as the output nodes, then the value only has to be propagated when the sign of the weight changes. The output nodes can then broadcast their δ values, and send a point to point message when the sign of a weight changes. The communication overhead of equation 5.5 is reduced when this modification is used.

Character Recognition
Propagation of Sign of the Weights

Algorithm Modification	Input Presentations	Execution Cycles
Limited Precision Standard	21,340	60.82M
Use Sign of Weight	27,510	56.26M

Table 5.5

Table 5.5 shows that there is a performance increase for the character recognition problem when the propagation of weights is reduced. An estimate has been made of how many clock cycles the target architecture requires to learn the application. The estimate is based on one clock cycle per parallel MAC operation, and one clock cycle for all other operations. Even though the network takes more input presentations to learn the data when only propagating the sign, there is less execution time. The reduction in execution time is caused by the fact that there are a lot of cycles required to back propagate the $\delta_i w_{ij}$ terms. As Table 5.6 indicates, the results for the coordinate transformation problem are not as good as the character recognition results.

Anderson's reasoning was that when the weights are small the error values for the hidden nodes will also be small. At the beginning of the training session the weights are normally set to small random values, so learning will be slow until the weights become large enough to have large error values for the hidden nodes. If the signs of the weights are used, the error values for the hidden nodes will be larger. This reasoning may be valid for simple problems, but more complex problems may require more resolution of the backward connection. The sign of the weight does not provide very much resolution for the hidden node error calculation. The magnitude of the weight is important when determining the contribution of one hidden node to an output node. If a hidden node is strongly connected to an output node that has a large error, then it should be reinforced more than a hidden node that has a weak connection to the same output node. If just the sign of the weights are used, then the hidden nodes will be reinforced equally if the sign of the weights are the same.

Coordinate Transformation
Propagation of Sign of the Weights

Algorithm Modification	Input Presentations	Execution Cycles
Limited Precision Standard	14,960	45.75M
Use Sign of Weight	209,900	499.00M

Table 5.6

As the simulation results in tables 5.5 and 5.6 suggest, Anderson's modification did not use fewer input presentations to learn the problems. In fact the learning time for the coordinate transformation simulation was much worse. The poor performance of the coordinate transformation simulation was due to the fact that the network was required to learn a continuous mapping. Using just the sign of the weight in equation 5.5 restricts the precision of the δ_i calculation. Because the coordinate transformation problem maps a continuous input space to a continuous output space, it needs more precision than the character recognition simulation. The algorithm seems to require more weight resolution than simply using the sign of the weight. If we use the entire value of the weight and propagate the weight only when it changes by a certain limit, then we can use greater weight precision and also use reduced communication. Tables 5.5 and 5.6 show the results of simulations that propagate the weights only when the weight differed by a certain threshold from the weight that is stored in the hidden nodes' local memories.

Only propagating weights that have changed by 0.1 has a significant increase in performance, and the coordinate transformation problem was also faster with a threshold of 0.3. Generally as the threshold increased, the number of input presentations required to learn the training set increased.

Character Recognition
Propagation of Weight After Threshold Change

Algorithm Modification	Input Presentations	Execution Cycles
Limited Precision Standard	21,340	60.82M
Threshold 0.1	21,050	43.06M
Threshold 0.3	38,780	79.31M

Table 5.7

The execution savings of reduced communication was lost when the higher threshold simulations required too many input presentations. The performance increase of using reduced weight propagation is application specific, and may not be beneficial in all circumstances. The additional cost of requiring more local memory for the hidden nodes must also be considered. Another solution would be to calculate the weight changes for the fan-out weights as well as the fan-in weights. Then the output nodes could broadcast the δ_k values and the hidden nodes would be able to have an up to date copy of the weights. The local memory would have to be large because momentum accumulations of the fan-out weights would also be stored in the hidden nodes' memories.

Coordinate Transformation
Propagation of Weight After Threshold Change

Algorithm Modification	Input Presentations	Execution Cycles
Limited Precision Standard	14,960	45.75M
Threshold 0.1	16,820	40.14M
Threshold 0.3	16,570	39.51M
Threshold 0.5	20,880	50.08M

Table 5.8

5.3. Sum Weight Changes

Accumulating the weight changes of equation 5.6 before updating the weights is a common modification to the back propagation algorithm. Weight change accumulation is similar to the data partitioning method used by Pomerleau, et. al. [18] when they optimized the back propagation algorithm for execution on the Warp system. Weight accumulation is also used in the conjugate-gradient optimization technique[19]. Le Cun [20] has suggested that adjusting the weights for every input presentation can create a stochastic path through the error surface. The ΔW_i for each input vector will have a different vector angle in the weight space. When the weight changes are accumulated, the path of the gradient descent algorithm tends to be less efficient if there is redundant information in the training set.

However, weight change accumulation can improve the execution time of the algorithm. The only value that needs to be accumulated from equation 5.6 is the $\delta_i x_j$ term. Multiplying the learning constant (η) and adding the momentum ($\alpha \Delta w_{ij}(t-1)$) can be delayed until the weights are actually updated. The communication of the error propagation will also be reduced because the $\delta_i w_{ij}$ values in equation 5.5 are only sent when the weights are actually updated. The simulation results in Table 5.9 and 5.10 support le Cun's intuition that the accumulation of weight changes requires more input presentations for the networks to learn. Although, the savings in execution time would be a motivation to use the weight accumulation modification. When evaluating the tradeoff between input presentations and execution time, it would be difficult to ignore the large savings in execution time that weight accumulation provides.

5.4. Noise

As previously discussed, an effect of using integer computation is that the weight space is quantized, and the weights can only be adjusted in discrete increments. It is possible for a weight to get stuck on one value if the weight changes are not big enough to bypass local minima. Adding random noise to the weights can help smooth the weight space. Modifying the

Character Recognition
Accumulated Weight Change

Algorithm Modification	Input Presentations	Execution Cycles
Limited Precision Standard	21,340	60.82M
Accumulate 2 Inputs	22,650	39.26M
Accumulate 10 Inputs	24,540	20.75M

Table 5.9

Coordinate Transformation
Accumulated Weight Change

Algorithm Modification	Input Presentations	Execution Cycles
Limited Precision Standard	14,960	45.57M
Accumulate 2 Inputs	15,390	33.65M
Accumulate 10 Inputs	18,080	26.96M

Table 5.10

weights by a small random value, may cause the network to bounce out of a local minima. Some applications of digital filters use noise to remove bias from the calculations. This digital filtering technique is called *dithering,* which randomly sets the least significant bit of a value. For the research presented in this thesis noise was inserted into the weight space by randomly setting the lower n ($1 \leq n \leq 4$) bits of each weight. It is counter-intuitive to think that a computation can be improved by noise, but the simulation results in Tables 5.11 and 5.12 suggest that the number of input presentations can be reduced.

The networks were even able to learn with four bits of random noise, which is less precision than the network needed without noise. The reason that the network was able to tolerate so much noise is that the Δw_{ij} values

Character Recognition
Effect of Noise on Learning.

Algorithm Modification	Input Presentations
Limited Precision Standard	21,340
1 Random Bit	18,250
4 Random Bits	25,800

Table 5.11

Coordinate Transformation
Effect of Noise on Learning.

Algorithm Modification	Input Presentations
Limited Precision Standard	14,960
1 Random Bit	10,890
4 Random Bits	12,100

Table 5.12

were accumulated without noise. The momentum would build up until the weight change was greater than the noise. Even though the network converges in fewer input presentations, the extra computation of adding the noise may increase the execution time of the algorithm.

CHAPTER 6

Category Learning

There is a class of pattern recognition algorithms that attempt to divide the input space into well defined regions. Each region will be represented by a different output node. The ANN model that will be analyzed in this chapter uses the algorithm that is described in [21]. The term *category learning* will be used for the model discussed in this chapter. Batchelor also published a very similar algorithm for pattern recognition[22]. Category learning is a supervised algorithm that maps an input space into a binary output space. The output layer should only have one node (class) on at any time. The output node that is on determines which region (category) of the input space the input vector is in.

The networks in the category learning model have three layers of nodes. The first layer presents the input vector to the network. For the category learning model, the input vector must be normalized to a unit length ($\|X\| = 1$). For a normalized input vector, the input space is the unit hypersphere (i.e. for a two dimensional input vector, all of the input points lie on a circle of radius 1). The middle layer contains nodes that detect when an input vector is within a particular region of the input space. The activation function of the prototype vector is a measure of how close the input vector is to the values of the weights of the prototype vector. The third layer has the output nodes that indicate which class (or category) the input vector is in. The weights between the prototype layer and the output layer only have the values of 1 and 0. Each node in the prototype layer is assigned to a class. The weight between a prototype and the output that represents the corresponding class has a value of 1, all other weights are 0.

Prototypes

The most important aspect of the category learning algorithm is the training of the prototype nodes. The activation function for the prototype nodes (g_i) is

$$g_i = \Theta(\Sigma f_j w_{ij}) \tag{6.1}$$

where f_j is the value for input j and w_{ij} is the connection strength between prototype i and input j. The linear thresholding function ($\Theta(x)$) is defined as

$$\Theta(x) = \begin{cases} 0 & \text{if } x \leq \theta \\ x-b & \text{if } x > \theta \end{cases} \tag{6.2}$$

where θ is a threshold constant that determines when the prototype will have a output value greater than 0, and b is a constant that modifies the output value.

The weights of the prototypes are modified when an incorrect classification is made. There are two types of incorrect classification. The first is when the output node that represents a category does not get activated at the proper time, which occurs when none of the prototypes that are assigned to the correct output node are activated. A new prototype is then created with the initial weights

$$w_{ij} = \lambda_i(0)p_{ij}, \;\; p_{ij} = f_j. \tag{6.3}$$

The initial weights of a prototype are set to the value of the current input vector times a scalar variable ($\lambda_0(t)$). The weights represent a region of the hypersphere that is centered on the original input vector (f_j), and with a radius that is determined by $\lambda_i(t)$ and θ. The variable $\lambda_i(t)$ lengthens the weight vector beyond the unit hypersphere, and determines the distance where the activation value for the prototype is zero. The dot product of an input and the weights of a prototype is equal to the cosine of the angle between the two vectors times $\lambda_i(t)$, so adjusting $\lambda_i(t)$ will vary the radius of the activation region for the prototype.

The second type of incorrect classification occurs when a prototype for the wrong class is activated. A prototype is activated incorrectly when the input vector for a different class is within the activation radius. The length variable is then modified by

$$\lambda_i(t+1)=\frac{1}{\Sigma p_{ij} f_j} \tag{6.4}$$

where f_j is the jth element of the input vector. The effect of modifying $\lambda_i(t)$ is that the incorrect input vector will lie on the radius of the prototype's region. With $\theta = 1$ in equation 6.2, the prototype will be activated for all input vectors that are closer than the last incorrectly classified input vector.

To understand how the category learning algorithm works, consider a network with three inputs. The normalized input space will lie on a three dimensional sphere. The first input vector that is presented will define a point on the surface of the sphere. Since no prototypes have been created, a new prototype will be put at the location of the input vector. The initial length variable ($\lambda_i(0)$) will define a circle on the surface of the sphere with the center at the prototype position. If the second input vector is assigned to the same category as the first prototype, and the position is within the circle, then a correct classification has been made and there will be no modification to the network. If the second input vector is inside the circle, but it is in a different category, then confusion occurs and the length variable of the first prototype will be reduced so that the second input does not lie within the circle. A new prototype will be allocated for the second input vector in this case of confusion. A new prototype will also be allocated if the second input vector is outside the circle of the first prototype.

As new input vectors are presented to the network, new prototypes will be created and old prototypes will be modified. Eventually the regions of the prototypes will cover the input space in such a way that correct classifications are made for all input vectors. One advantage of the classification model is that learning is much quicker than with gradient methods such as back propagation. A disadvantage is that it may take many prototypes to accurately map the regions of the input space, especially if the regions are close together. Another disadvantage of the category learning algorithm is that it may not be fault tolerant if implemented in hardware. Each prototype defines a distinct region of the input space. So if one prototype fails, the network will not generalize well for the region that was covered by the faulty prototype. The prototypes have a *localized* representation of the input space as opposed to the *distributed* representation of the back propagation network.

The training set used for the simulations of the category learning model was the character recognition problem that was described in the previous chapter (figure 5.1). The inputs were a linear representation of the 12×12 characters, but for this simulation the input vector was normalized. There were twenty six output classes, one class for each character. The input and output layers are the same size as in the back propagation solution to the character recognition problem. New prototypes were created until the network could correctly classify all 286 characters. The simulations were considered complete when a complete epoch was presented without creating a new prototype and without confusion. A complete epoch must be run after each network modification because a new prototype may have an input vector of another class within the default range ($\lambda_i(0)$). Conversely, if the range of a prototype contains more than one input vector for the correct class, and the range is reduced because of confusion, then the new range may exclude a previously classified input vector.

6.1. Limited Precision

The precision limit of the category learning algorithm is different for each application. Quantization of the normalized inputs causes the unit hypersphere to have discrete steps. The limit of quantization is due to the resolution of the angle between vectors of different classes. If the angle between two vectors is close enough so that the dot product between the vectors is less than the angle resolution, then the two vectors cannot be differentiated. The precision limit for an application is dependent on the number of inputs, the distance between vectors of different classes, and the distribution of the training set in the input space.

To visualize the quantization of a hypersphere, imagine a globe that has a grid line for every degree of longitude and latitude, and cities on the globe are positioned with the resolution of a single degree. If two cities are closer together than the resolution of the grid, then they will both be mapped to the same point on the grid. Perhaps Beaverton and Hillsboro would be given the same position on the globe. If the precision of the map is increased, then the two cities would be able to have unique locations. So in this case, the precision required to map the problem is dependent on the distance between the two closest input vectors that are in different classes.

To describe the effects of the input space on the precision limit the character recognition problem will be used as an example. The input vectors are binary vectors that have been normalized. Conceptually it is a hypercube (the binary vector) that has been mapped onto a hypersphere (the normalized vector). The resolution of the hypersphere may cause two neighboring corners of the hypercube to be mapped to the same vector. Then the category learning algorithm can not tell the difference between two characters that differ by a single pixel. The mapping of the hypercube to the hypersphere is only incorrect if vectors of different classes are mapped to the same location. If the two vectors belong to the same class, then there will not be an incorrect categorization.

For applications that have a binary input space, the precision is also dependent on the number of inputs that can be on in the input vector (on = 1, off = 0). So the resolution of the calculations is dependent on the domain of the training set. To illustrate this point, consider a binary vector of arbitrary length. If only one bit is on at any time, then the normalized vector will be identical to the binary vector. The input vector will be at the intersection of the axis of the input dimension on the hypersphere. Only one bit of resolution is required, since the normalized inputs are binary. If up to two bits of the binary vector can be on, then the normalized vector can be at the midpoint of an arc between two of the axes on the hypersphere.[1] Two bits are required to distinguish between vectors that can have up to two input values on. The number of bits of precision required for an application increases logarithmically with the number of input values that can be on. Three bits of precision will be able to distinguish between vectors with up to four bits on. If i is the number of bits that can be on in an input vector, then the normalized vector requires $\lceil \log_2(i) \rceil + 1$ bits of precision.

Since the character recognition problem has 144 inputs, then the category learning algorithm could conceivably need nine bits of precision. However, there are no characters with more than 55 bits on, so six bits of precision is sufficient. Although the worst case precision can be calculated, more precision could allow better results because the category learning algorithm would be able to set λ with better resolution. A single prototype would be able to contain more than one input vector of the appropriate class. For this reason, eight bit weights were used for the simulations. For real valued input vectors that are being normalized, the required precision is completely dependent on the training set. As mentioned earlier, the distance

[1] Note that the input of a binary vector that is normalized is only in the positive quadrant of the hypersphere. The zero vector is at the center of the hypersphere, and is not a legal input.

between the two closest vectors that are in different classes will determine the amount of resolution that is required for an application.

6.2. Threshold Adjustment

One potential problem with implementing an architecture for the category learning algorithm is that equation 6.4 requires a divide operation. The hardware for fast addition, subtraction, and multiplication operations can be easily implemented. The divide operation, however, is relatively slow in execution speed. Fast divisions may be possible, but not with the limited silicon area of the target processors that are used in this research. Elimination of the division calculation in equation 6.4 would speed the prototype learning. Fortunately, the division operation can be removed with a little algebra.

In the calculation of the output values for the prototype layer (equations 6.2, 6.3, and 6.4), the threshold (θ) is kept constant. If θ is modified during learning, instead of the weight variable (λ), then the radius of the range of a prototype can be adjusted without using a division operation. The λ term would be made a constant, and θ would be the variable that is modified for adjusting the radius of the prototype. With $\lambda = 1$, the weight matrix can be fixed to the value of the input vector that was present when the prototype was created. So equation 6.3 is rewritten to be

$$w_{ij} = p_{ij} = f_j. \tag{6.5}$$

Each prototype is given a unique $\theta_i(t)$ for modifying the radius of the region categorized. When an input vector is incorrectly classified by a prototype, then the radius is adjusted by the equation

$$\theta_i(t+1) = \Sigma f_j w_{ij} \tag{6.6}$$

When the incorrect input is presented again, the vector will be on the border of the radius of the prototype and the output value of the prototype will be 0 (equation 6.2). The result of removing the weight variable λ and modifying the threshold constant θ is that the network performs the same as with original equations, except that there is a potential for faster learning and less silicon area required for the architecture.

6.3. Binary Input Vectors

One part of the category learning algorithm that makes hardware implementation difficult is the requirement for normalized inputs. ANN algorithms use normalized inputs to keep the input vector on the unit hypersphere. All vectors that are on a hypersphere have the same length, which is the distance between the origin and the surface of the hypersphere. The advantage of having all input vectors the same length is that two vectors can be compared by calculating the cosine of the angle between the two vectors. The calculation for normalizing a vector is

$$f_i = \frac{x_i}{\left(\sum_{j=0}^{n} x_j^2\right)^{\frac{1}{2}}} \tag{6.7}$$

Where f_i is the normalized value of the original value x_i. A disadvantage of an algorithm that uses normalized inputs is that the normalization is computationally expensive, and it is not easily parallelizable.

It is not as effective to normalize binary vectors as it is to normalize real vectors. It would be desirable to be able to compare binary vectors without normalization. As discussed above, a binary input space defines a hypercube. Each binary input vector is located at a corner of the hypercube. For binary vectors, we are no longer comparing the angle between two different vectors on a hypersphere. Instead we are measuring the distance between two corners on the hypercube by traversing the edges. The distance between two vectors is equal to the number of values that are different.

The distance between binary vectors is often called the *hamming distance.* However, if the inputs of the binary input vectors have the domain of (0,1) equation 6.1 will not provide an accurate distance between two different vectors. Consider an unnormalized binary input space with three inputs. If a prototype was created for the input (1,0,0), then the prototype would not be able to distinguish between any other input vector that had a value of 1 for the first element. The weights of the prototype that have a value of 0 do not contribute to the activation value, so the prototype would detect a perfect match for input vectors that have the values of {(1,0,0),(1,1,0),(1,0,1),(1,1,1)}. Another example, taken from the character recognition problem, is that in many of the fonts the only difference between an O and a Q is the tail of the Q. When the Q is presented as an input, the prototypes for the O will be incorrectly activated.

The category learning algorithm will work if the binary input vectors (and prototype weights) have a value of (−1,+1). A binary vector with values of (−1,+1) is called *bipolar*. The result of equation 6.1 using bipolar vectors is equal to

$$g_i = \Theta(\Sigma f_j w_{ij}) = \Theta(N - 2H(F, W_i)) \tag{6.8}$$

where F is the input vector, W_i is the weight vector for prototype i, N is the dimension of the input space, and $H(F, W_i)$ is the hamming distance between the input vector and the weight vector. The output value of a prototype that perfectly matches an input vector is equal to the number of values (dimensions) in the input vector. For an input vector that is not a perfect match, the output value of the prototype is equal to the number of values in the input vector minus two times the number of bits that differ.

Using bipolar vectors, the O and the Q characters can be distinguished without normalizing the input vectors. The weights can be represented with only one bit of precision by converting the 0 bits into a -1 before the calculation. The precision required for the output values is $\log_2(N)+1$, where N is the number of bits in the input vector. The precision limit is caused by the fact that the range of the output of a prototype is $(-N \leq g_i \leq N)$. The prototype will have a value of N for a perfect match, and a value of $-N$ for an input vector that has no elements in common. An advantage of using binary vectors is that fast hardware can be small and inexpensive.

Table 6.1 shows the results of the category learning simulations using floating point, eight bit integer, and bipolar representations. The simulations were run until all input vectors were correctly classified, and there were no incorrect classifications. The values shown are the number of prototypes that the simulations used for each category (character). Some categories, such as the categories for the letter C needed one prototype per font, while the T category only needed five prototypes to perform correct classification. The character recognition algorithm is a difficult problem for the category learning algorithm because some of the input vectors that are not in the same class have only two pixels that are different. As Table 6.1 shows, the eight bit integer simulation only needed one more prototype, and the unnormalized bipolar simulation needed seven extra prototypes to learn the problem.

Category Learning.
11 Font Character Recognition Problem.

Character	Float	Integer	Bipolar
A	6	6	8
B	10	10	10
C	11	11	11
D	11	11	11
E	8	8	8
F	8	8	8
G	11	11	11
H	10	10	11
I	6	6	6
J	10	10	9
K	7	7	10
L	7	8	8
M	7	7	8
N	9	9	10
O	11	11	11
P	11	11	11
Q	11	11	11
R	11	11	11
S	8	8	6
T	5	5	5
U	9	9	9
V	8	8	8
W	10	10	10
X	8	8	9
Y	7	7	7
Z	6	6	6
Total	227	228	234

Table 6.1

CHAPTER 7

Kohonen Maps

The last ANN algorithm that will be discussed is Kohonen maps. Kohonen maps were originally proposed by Tuevo Kohonen [23]. The Kohonen map is a type of unsupervised learning algorithm. Unsupervised algorithms do not use a training input to modify the weights. Without any training signal to use as a measure of correct performance, most unsupervised algorithms minimize an internally generated error measure to extract features of the input space. Unsupervised networks are often used to create a representation of the the input vector that has a lower dimension than the input space. The Kohonen maps' learning algorithm attempts to modify the weights so that the weight space accurately represents the distribution of the input training vectors in the input space.

For many applications, the correct input/output mapping may not be known. In cases where a training signal is not present, unsupervised algorithms may be desirable. An unsupervised algorithm can categorize input vectors based on the position within the input space. However, an unsupervised algorithm can not detect features such as recognizing the the difference between an 'a' and a 'b' with multiple fonts. For this reason, unsupervised networks are sometimes used in the first layer of a multilayer ANN system [4].

Kohonen networks typically have two layers, the input layer and the output layer. The input layer presents the input vector unchanged, and the output layer is fully connected to the input layer. The weights of a node in the Kohonen network define a point in the input space. The calculation of the activation values of a Kohonen map differs from most ANN models because it is not based on the linear combiner. The output value of a node is the Euclidean distance between the input vector and the weight vector.

$$o_i = \left[\sum_{j=0}^{N-1} (x_j - w_{ij})^2 \right]^{\frac{1}{2}} \tag{7.1}$$

Where o_i is the output value of node i, x_j is a value in the input j, and w_{ij} is the weight connecting input j and node i. Equation 7.1 is the geometric equation that defines the distance between two points in Euclidean space, so the node that is closest to the input vector will have the lowest output value.

The nodes in a Kohonen network have a fixed ordering. The ordering of the nodes in a network can be defined by an undirected graph of arbitrary dimension. Edges between two nodes in a graph define immediate neighbors. The organization of a layer can be any dimension. A one dimension layer has the nodes organized in a linear network, and each node $\{i\}$ that is not an endpoint has two neighbors $\{i-1\}$, and $\{i+1\}$. A two dimensional layer has the nodes organized in a mesh, and each non-border node $\{m,n\}$ has four neighbors $\{m-1,n\}$, $\{m+1,n\}$, $\{m,n-1\}$, and $\{m,n+1\}$.

The learning rule iteratively moves the weights of a node closer to the input vector. The equation for modifying the weights is given by the equation

$$\Delta w_{ij} = \alpha(x_j - w_{ij})N_{c.} \tag{7.2}$$

Where α is the learning rate, and N_c defines which nodes will have the weights modified. If node y is the node that is closest to the input vector $(min(o_i))$, then the value of N_c is

$$N_c = \begin{cases} 1 & \text{if } |y-i| < \eta \\ 0 & \text{if } |y-i| \geq \eta \end{cases} \tag{7.3}$$

Where η is the neighborhood variable. Only the nodes within a distance of η from the node o_y modify their weights. The weight update equation (7.2) moves the point defined by the weights of a node closer to the current input vector.

After a training session, the nodes in a Kohonen map should be spread across the input space. Equation 7.2 moves the weights of a node towards the input vectors. Equation 7.3 allows the closest node to pull along the neighboring nodes, while the nodes that are farther away remain in other areas of the input space. The concentration of nodes on the input space depends on the input distribution of the training set. If the input vectors are evenly distributed, then the Kohonen map will be evenly spread across the input space. If there is not an even distribution of input vectors, then the Kohonen map will have a heavier concentration of nodes in the areas where there are more input vectors.

To understand how a Kohonen map works, consider a network that has a single node. Since there is only one node, it will always be the closest to the input vector and it will always have its weights modified. Each weight update will move the point in the input space that is defined by the weight vector closer to the current input. The effect of equation 7.2 over time is to move the node toward the mean of the input distribution (not the center of the input space). However, if the learning rate (α) is too large, then the node will be disproportionately closer to the last input vectors presented. In fact, if ($\alpha = 1.0$) then the weights of the node will be equal to the last input vector presented to the network.

For a network, the behavior of each node is somewhat different than with just a single node. Early in the training phase, the network will expand rapidly. For each input presentation, the closest node and its neighbors will move towards the input vector. With a high value for α and η, the nodes will jump around in the input space. However, the topology of the network will cause the network to take the rough shape of the input distribution. As the learning variables are reduced, the node movement will slow down, and each node in the network will move toward the mean of a region in the input distribution. When η is reduced to zero, each node will only be influenced by a small subset of the input vectors, and the nodes will move to the center of a subspace.

Figure 7.1 shows the position of the nodes in a Kohonen network at different times during the training session. The input vectors have two dimensions that are random values between 0 and 1. The network is two dimensional and has sixty four nodes. The topology of the network is a mesh that has eight nodes in each direction. Lines are drawn between neighboring nodes. In a typical training session, the learning rate (α) and

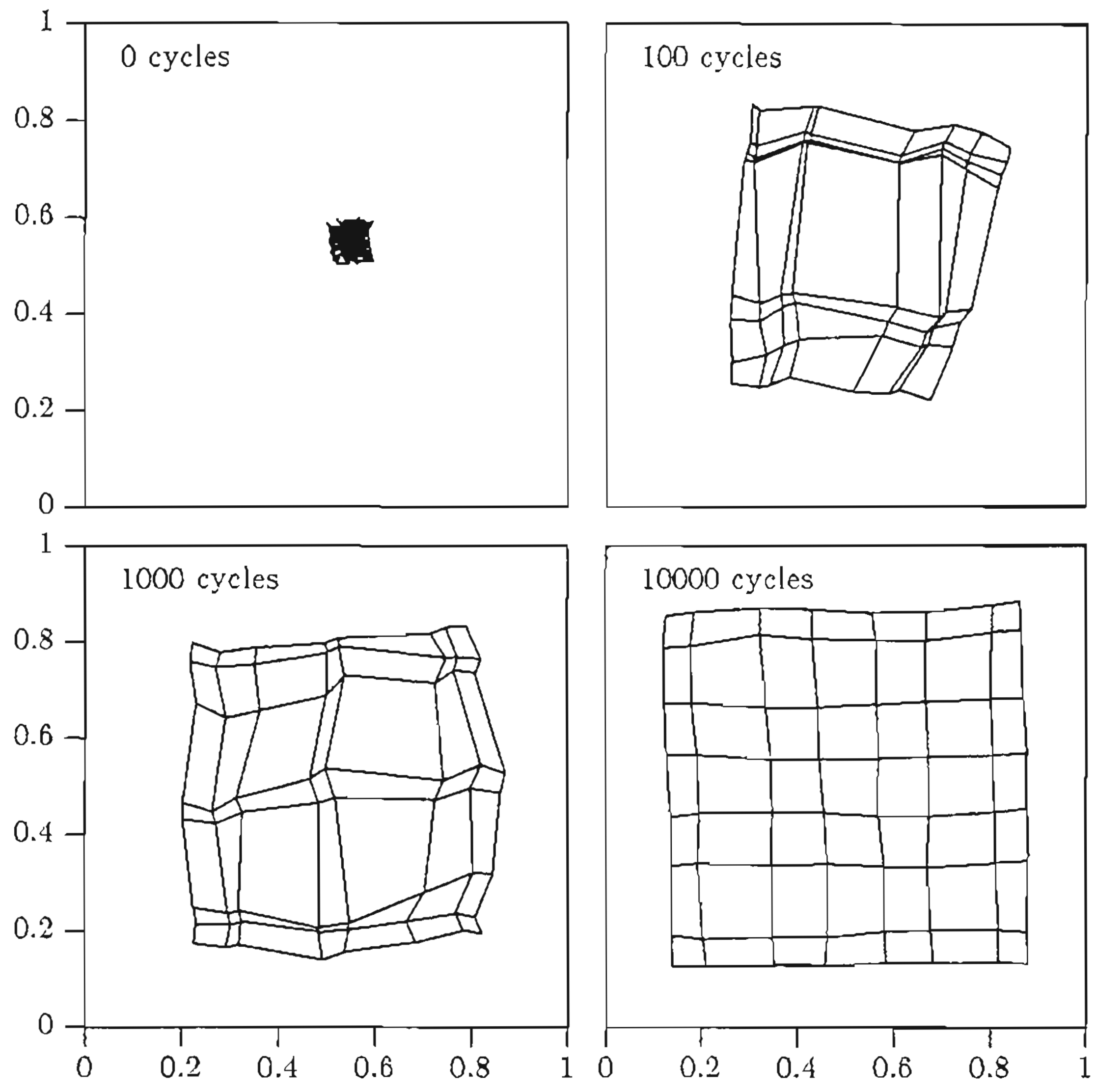

Figure 7.1 - Expansion of a Kohonen map.

the neighborhood (η) are kept relatively large so that the network will spread out quickly. As training proceeds, the learning variables are reduced.

The neighborhood (η) is normally set to 0 near the end of the training session so that each node will move towards the center of a small region of the input space. The learning rate (α) is set to a small value (≈ 0.01) so that the nodes will be at the mean of the local region, and not be unfairly influenced by the last few inputs.

Figure 7.2 shows the network after it has been trained with different input shapes. In 7.2(a) the input vectors are randomly distributed within a

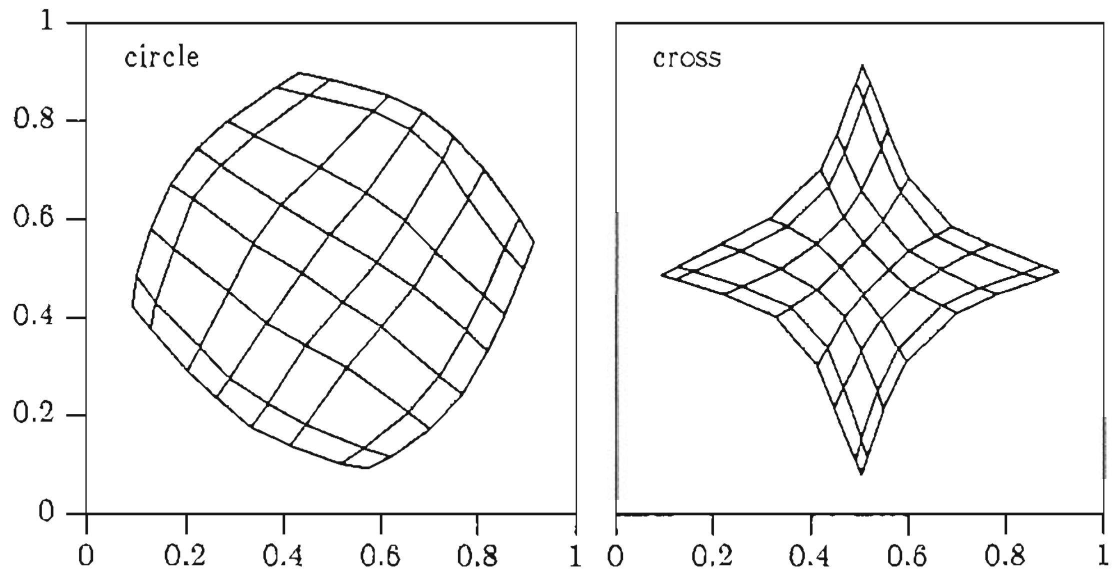

Figure 7.2 - The Kohonen map network on different input spaces.

circle, and in 7.2(b) the input vectors are randomly distributed within a cross. In both simulations that are shown in 7.2, the network has spread out in the input space.

Figure 7.3 shows a one dimension network that has been trained on different input distributions. As in the first simulation, the domain of the inputs is (0,1). Figure 7.3a shows a network that has been trained on an even input distribution. Figure 7.3b shows a network that has been trained

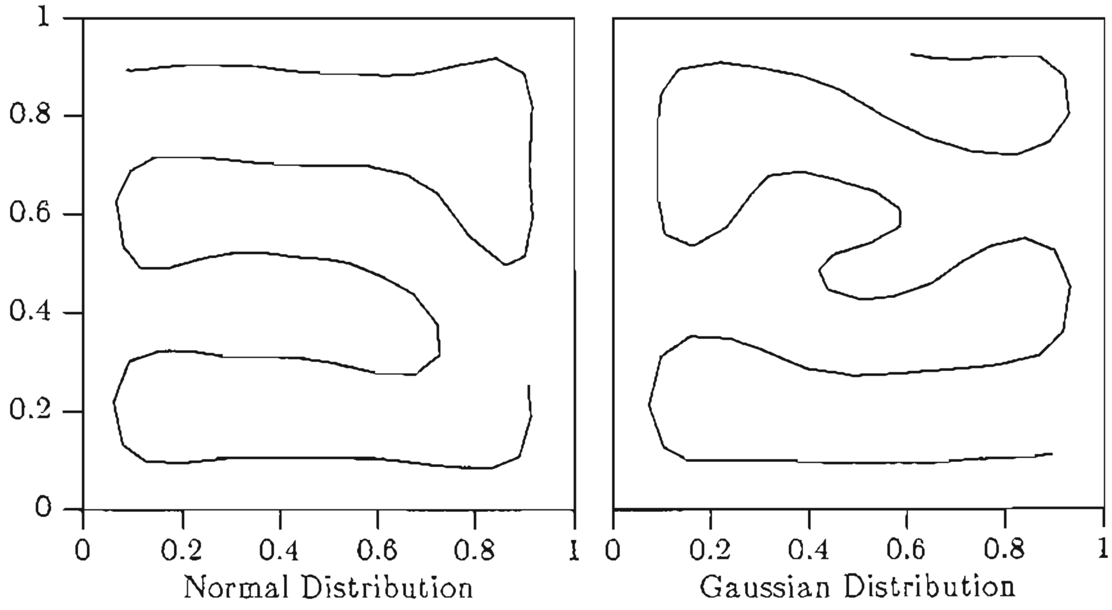

Figure 7.3 - A Kohonen map network on different input distributions.

on an input space that has the same domain, but the input vectors have a higher probability of being in the center of the input space than on the outside. The network has learned the input distribution and is more concentrated towards the center of the input space.

7.1. Activation Function

One part of the Kohonen map algorithm that would be hard to implement in silicon is the calculation of the output values (equation 7.1). As discussed in the previous chapter, it is difficult to design a fast and small square root function. Fortunately, several researchers [24] [25] have found that it is not necessary to actually do the square root calculation for the Kohonen maps algorithm.

The purpose of the output values is to find the node with the weight vector that is closest to the input vector. Since the square root function is a monotonically increasing function, it does not affect the distance comparison between nodes of a network. The node with the weight vector that is closest to the input vector will have the lowest value before and after the square root calculation is made. Equation 7.1 can be rewritten in vector form as

$$o_i = (X - W_i)^2 = X^2 - 2XW_i + W_i^2 \tag{7.4}$$

Equation 7.4 can be further reduced by removing the X^2 term since it is a constant for all outputs (o_i).

$$o_i = W_i^2 - 2XW_i \qquad (7.5)$$

A possible speedup in the implementation of equation 7.5 is to calculate the (W_i^2) term only when the weights are modified. This reduces the output calculations to a constant (W_i^2) minus the two times the linear combiner function ($2XW_i$). However, with a perfectly parallel system (one CN per PN), there is no speedup with the last optimization because the processors that are not modifying weights will be idle while the other processors are calculating the square of the weights.

7.2. Limited Precision

The limits of precision for Kohonen maps are dependent on the domain of the inputs, the size of the network, and the weight modification calculation. If the dynamic range of the input values is large, then the weights of the network may require a high precision to determine the shape of the input distribution. However, if there are not very many nodes in the network, then the calculations may do well with less precision. As an example, consider again the degenerate case of a network with only one node. If the input values are evenly distributed with thirty-two bits of precision, the node should be able to find the approximate center of the input space with only sixteen bits of precision. The sixteen significant bits can have zero padding to the right, or the size of the input values can be reduced. A network that only has a few nodes should also be able to learn the shape of the input distribution without requiring much precision. A large network will require more precision, for reasons similar to that of the category learning algorithm. If the region of input space that each node maps is small, then the network will require more precision to distinguish between nodes. If the resolution of the weight space is not enough to tell two nodes apart, then the input space will not be correctly mapped.

Another important limit to the precision of the Kohonen map algorithm is the weight modification algorithm (equation 7.2). If the learning rate (α) is too high, then the last few input vectors may have too much of an influence on the network. However, a small learning rate creates a small weight modification. The distance value that the learning rate is multiplied by in equation 7.2 ($x_j - w_{ij}$) is usually small too, because the nodes that are modified have weight vectors that are close to the input vector. So the modifications that are made to the weights are normally small values. If there is not enough precision then the weights will not change, because the change in weights is a smaller value than the smallest quantity that can be stored. For the example presented in this chapter, eight bits of precision for the weights was not enough when η=0.01. The weight modifications are less

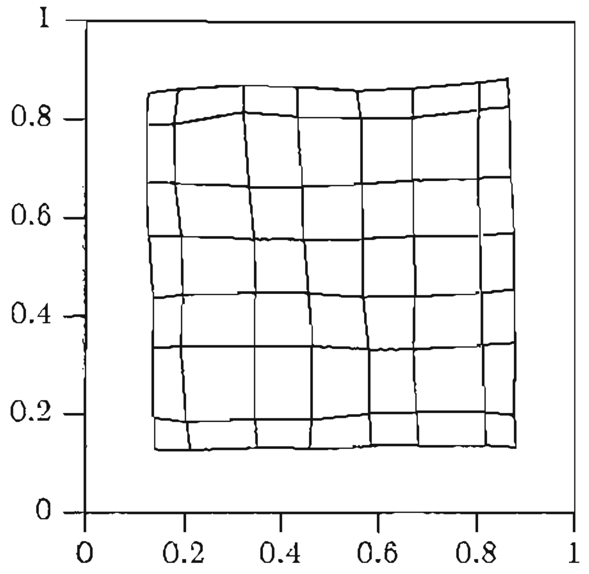

Figure 7.4 - A Kohonen map using sixteen bit weights.

than the lowest quantity that an eight bit value can have (≈0.004).

Figure 7.4 indicates that sixteen bits of accuracy was adequate for this example. With eight bit weights, the network would look like one of the early frames of Figure 7.1. The weights would stop learning as the learning rate (α) is reduced. Other input training sets may require more precision depending on the range of the values, and the number of nodes in the network.

CHAPTER 8

Conclusions

This research has provided a look inside three artificial neural network algorithms. An attempt was made to explore some of the implementation limits of the algorithms, and the reasons for the limits. In most cases, successful simulations showed that the algorithms could be implemented efficiently. Even though there were some simulations that were not successful, the knowledge gained from the failures was as valuable as the successes.

The results of the research described in this thesis indicate that:

(1) Limited precision calculations can be used for the three ANN algorithms simulated.

(2) The communication overhead of the back propagation algorithm can be reduced.

(3) The execution time of the back propagation algorithm can be reduced by summing the weight changes over several input presentations, even though more input presentations are required for convergence.

(4) Introducing noise into the weight space can help decrease the number of input presentations that is required for the back propagation algorithm to converge.

(5) The category learning algorithm can be calculated without the divide operation.

(6) If the category learning algorithm is given a binary input space, then binary weights can be used.

(7) The calculations for the distance measure used for the Kohonen self organizing maps algorithm can be significantly reduced.

Perhaps the most drastic (and the most important) behavior modification, is the use of limited precision calculations. The simulations showed that all of the algorithms used in this research could be implemented with limited precision, fixed point integer calculations. The actual limits of precision are not clearly defined. It is best to err on the side of too much precision, rather than risking a failure to implement an application. However, for all of the algorithms, the simulations indicate that reasonable hardware implementations can be made.

Some of the difficulties in implementing the algorithms with hardware were discussed. In most cases satisfactory solutions were found. Several of the solutions had equivalent results to the original algorithms, these were the most satisfactory because they promised a performance increase while the original theory was preserved. Other solutions modified the behavior of the algorithms in order to avoid difficulties. The benefits of the performance increase of the latter modifications must be compared with the possible negative effects of the behavior differences.

One of the least visible accomplishments of this research is the simulator that has been programmed for ANN experiments. Without this simulator, the amount of programming and computation time would have prohibited the research. Publicly available ANN simulators are normally not designed for fast execution and easy programming. In fact, many simulators are not designed for algorithm modification. With the CAPsim simulators almost all of the algorithm modifications are selectable from the user interface, the only exception is the limited precision computation. Execution of the simulation would be too slow if the data type was selectable at run time. A compiler switch was used to determine the computation that was used in the simulator. So all of the modifications to the simulators were selectable without changing the source code.

When considering the modifications presented in this thesis it is important to consider the tradeoffs that must be made. The benefits of the algorithm modifications are dependent on how they are applied. For all of the algorithms, applications can be found that require extreme precision for success. However, knowledge of the limits of the algorithm will help the designer to create working applications.

Although the results are promising, the real test will come when the algorithms are used for real applications on actual hardware. This research should be beneficial to hardware architects that are designing ANN systems. The analysis should also be a help to those who want to create working applications based on the ANN algorithms given adequate hardware.

There is much research left to be done. The work presented in this thesis is in no way a comprehensive analysis of ANN algorithms. There are many more algorithms that deserve similar treatment. In fact, there are many more modifications that may be useful for the algorithms that are presented in this paper. The simulations were designed to stay as close to the original algorithms as possible when modifications were made. Some of the algorithms may benefit from more drastic modifications. Other simulations could test the behavior of the algorithms when implemented with analog computation. And finally, the ultimate research that must be done is to implement the algorithms on parallel hardware that is designed to execute ANN algorithms.

Bibliography

[1] Bailey, Hammerstrom (1986). How to Make a Billion Connections. Oregon Graduate Center T.R. #CS/E-86-007. Beaverton, OR.

[2] Widrow, Stearns (1985). *Adaptive Signal Processing.* ISBN 0-13-004029 01, Prentice Hall.

[3] Hornik, Stinchcombe, and White (1989). Multilayer Feedforward Networks are Universal Approximators. *Neural Networks, Volume 2, Number 5, 1989.* Pergamon Press.

[4] Leen, T.K. and Rudnick M., "Unsupervised learning for improved supervised classification," *In preparation.*

[5] Bahr, Bailey, Baker, Hammerstrom, Jagla, Johnson, Mates, May, McCartor, Means, Rudnick (1988). The OGC Cognitive Architecture Project: Silicon Implementation of Connectionist/Neural Networks. *NorthCon 88.* Seattle, WA.

[6] Blelloch, G., Rosenberg C.R. (1987). Network learning on the Connection Machine. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence,* Milan, Italy.

[7] Holler, Tam, Castro, Benson (1989). An Electrically Trainable Artificial Neural Network (ETANN) with 10240 "Floating Gate" Synapses. *International Joint Convergence on Neural Networks.* Washington D.C.

[8] Bailey, Hammerstrom (1988). Why VLSI Implementations of Associative VLCNs Require Connection Multiplexing. *International Conference on Neural Networks.* San Diego, CA.

[9] Rudnick (1988). Physical Broadcast Structure. Oregon Graduate Center T.R. #CS/E-88-018. Beaverton, OR.

[10] Werbos (1974). PhD. Dissertation. *Beyond Regression.* Harvard University.

[11] Parker (1985). Learning Logic. Massachusetts Institute of Technology T.R. #TR-47.

[12] Rumelhart, Hinton, Williams (1986). Learning Internal Representations by Error Propagation. In D.E. Rumelhart & J.L. McClelland's (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition (Vol. 1)* (pp. 318-362). Cambridge, Mass.: MIT Press.

[13] Vogl et al., Accelerating the Convergence of the Back-Propagation Method. *Biological Cybernetics, Vol. 59, pp 257-263, 1989.*

[14] Parker (1987). Optimal Algorithms for Adaptive Networks: Second Order Direct Propagation, and Second Order Hebbian Learning. *IEEE First International Conference on Neural Networks.* San Diego, CA.

[15] Becker, le Cun (1988). Improving the Convergence of Back-Propagation Learning with Second Order Methods. Department of Computer Science, University of Toronto. Technical Report CRG-TR-88-5. Toronto, CANADA.

[16] Sejnowski, Rosenberg (1986). NETtalk: A Parallel Network that Learns to Read Aloud. The Johns Hopkins University Electrical Engineering and Computer Science Technical Report JHU/EECS-86/01.

[17] CA Anderson, Ph.D. Dissertation, *Learning and Problem Solving with Multilayer Connectionist Systems,* Amherst, MA, Sept. 1986.

[18] Pomerleau, Gusciora, Touretsky and Kung (1988). Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. *International Conference on Neural Networks.* San Diego, CA.

[19] Barnard, Cole (1989). A neural-net training program based on conjugate-gradient optimization. Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology. Technical Report No. CSE 89-014.

[20] le Cun (1989). Generalization and Network Design Strategies. University of Toronto T.R. #CRG-TR-89-4. Toronto, CANADA.

[21] Reilly, Cooper, Elbaum (1982). A Neural Model for Category Learning. *Biological Cybernetics.* Springer-Verlag.

[22] Batchelor (1978). *Pattern Recognition, Ideas in Practice.* Plenum Press. New York and London.

[23] Kohonen (1984). Self Organizing Feature Maps. *Self-Organization and Associative Memory.* Springer-Verlag.

[24] Winters, Rose (1989). Minimum Distance Automata in Parallel Networks for Optimum Classification. *Neural Networks, Volume 2, Number 2, 1989.* Pergamon Press.

[25] Nilsson (1965). *Learning Machines.* McGraw-Hill Book Company.

Biographical Note

The author was born in Brawley, California on April 10, 1958. He graduated from North Bend High School in 1976, and received a B.S. in Computer Science from Oregon State University in 1980. He worked for several years designing industrial process control systems before attending the Oregon Graduate Institute of Science and Technology. He is currently employed by Adaptive Solutions Inc. of Beaverton, Oregon. His professional interests are neural networks, parallel processing, image processing and computer graphics.