Iowa State University
Digital Repository

Retrospective Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

1990

Neural networks: genetic-based learning, network architecture, and applications to nondestructive evaluation

James M. Mann
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Engineering Commons

Recommended Citation
Mann, James M., "Neural networks: genetic-based learning, network architecture, and applications to nondestructive evaluation" (1990). *Retrospective Theses and Dissertations*. 241.
https://lib.dr.iastate.edu/rtd/241

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

المنارة للاستشارات

www.manaraa.com

Neural networks: Genetic-based learning, network architecture, and applications to nondestructive evaluation

by

James Michael Mann

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

In Charge of Major Work

For the Major Department

For the Graduate College

Iowa State University
Ames, Iowa

1990

TABLE OF CONTENTS

1 INTRODUCTION 1

2 NEURAL NETWORKS 4
- 2.1 Overview 4
- 2.2 History 6

3 BACKPROPAGATION LEARNING IN FEED-FORWARD NEURAL NETWORKS 12
- 3.1 Overview 12
- 3.2 Feed-Forward Neural Networks 13
- 3.3 Determining a Neurode's Output 14
- 3.4 Supervised Learning 17
- 3.5 The Backpropagation Learning Algorithm 18
- 3.6 Activation Functions 24
- 3.7 Advantages and Disadvantages of Backpropagation Learning 27

4 INVERSION OF UNIFORM FIELD EDDY CURRENT DATA BY USING NEURAL NETWORKS 30
- 4.1 Overview 30
- 4.2 Introduction 31
- 4.3 Uniform Field Eddy Current Probe 31
- 4.4 Approach 34
- 4.5 Implementation 35
- 4.6 Results Using Synthetic Data 36
- 4.7 Results Using Experimental Data 39
- 4.8 Discussion 45

4.9 Conclusions 46

5 GENETIC-BASED LEARNING IN ARTIFICIAL NEURAL NETWORKS 48

5.1 Genetic Algorithms 48

5.1.1 Introduction 48

5.1.2 Genetic algorithm terminology 49

5.1.3 Genetic algorithm notation 52

5.1.4 Schemata 52

5.1.5 Genetic algorithm operators 57

5.1.6 Building blocks 63

5.2 Application of Genetic Algorithms to the Optimization of Weights in Neural Networks 71

5.2.1 Chromosomes 71

5.2.2 Genotypes 73

5.2.3 Population 73

5.2.4 Algorithm 74

5.2.5 Genetic diversity 74

5.2.6 Parameters 76

5.2.7 Fitness value 79

5.3 Genetic-Based Learning Enhancements 80

5.3.1 Elitist model 80

5.3.2 Sliding window to induce selective pressure 81

5.3.3 Adaptive mutation operators 83

5.3.4 Parallel implementation 90

5.4 Conclusions 92

6 COMPARISON OF BACKPROPAGATION AND GENETIC-BASED LEARNING 94

6.1 Overview 94

6.2 Computational Requirements of the Backpropagation Learning Algorithm 94

6.3 Computational Requirements of the Genetic-Based Learning Algorithm 97

6.4 Implementation 103

6.5 Comparison of Backpropagation and Genetic-Based Learning Times 105

6.5.1 Selection of the mutation probabilities 106
6.5.2 Details of the learning comparisons 107
6.5.3 Parity M 109
6.5.4 Symmetry M 113
6.5.5 Encoder M 115
6.5.6 Adder 2 116
6.6 Conclusions and Discussion 118

7 NEURAL NETWORK ARCHITECTURE CONSIDERATIONS 121
7.1 Overview 121
7.2 Motivation 123
7.3 Background 126
7.4 Feature Detection 127
7.5 Dynamic Neurode Creation 131
7.6 Feature Detection Using Dynamic Neurode Creation 134
7.6.1 XOR feature detection 135
7.7 Conclusions and Discussion 143

8 SUMMARY AND DISCUSSION 146
8.1 Overview 146
8.2 Summary 146
8.2.1 Inversion of uniform field eddy current data 146
8.2.2 Genetic-based learning 147
8.2.3 Network architecture 148
8.3 Discussion and Suggestions for Future Work 149
8.3.1 Inversion of uniform field eddy current data 149
8.3.2 Genetic-based learning 150
8.3.3 Network architecture 152

9 BIBLIOGRAPHY 153

10 ACKNOWLEDGEMENTS 158

1 INTRODUCTION

Even before the invention of the electronic digital computer by John Vincent Atanasoff [1], there developed a desire to create a machine which could emulate the functionality and thought processing of a living organism. This quest initially began as a movement called "cybernetics" in the 1940s and was later formalized and coined Artificial Intelligence (AI) by researchers in the mid-1950s [2]. Since that time the majority of advances in the field of AI have been related to symbolic processing of information using expert systems. In such systems, a set of rules is developed which can be manipulated by a program, or inference engine, in order to draw conclusions from a given set of input data, often in conjunction with a data base of facts, to produce an appropriate response. While this method has proven effective in many situations [3,4], the process relies on the ability to create formal representations of the problem at hand and generate a set of rules that appropriately describes the interaction of these representations. Rules are usually generated through consultation with a human expert by a knowledge engineer (the person implementing the expert system). Many refinements of the rules are often necessary, requiring many interviews and adjustments to the expert system. This can involve hundreds or thousands of person-hours. Furthermore, these types of systems are often sensitive to noise in the input data--a slight perturbance can produce significantly different results. Also, expert systems can require hundreds or thousands of inferences for a given set of input data, thus requiring great computational power. It is clear that this is not the sort of processing which occurs in biological organisms, as the neuronal elements comprising these organisms are typically orders of magnitude slower than their electronic, artificial

counterparts. These underlying assumptions and limitations make it extremely difficult, if not impossible, to produce expert systems which perform tasks nearly every human performs routinely, such as reading hand-written text.

This would imply there are many such tasks which can not be performed efficiently by a computer simply by providing it with a set of rules and a collection of facts. Therefore, it seems necessary to develop mechanisms by which a system can learn by example and through experience--the same way humans learn. By repeatedly presenting a set of input conditions and monitoring the associated response, the system parameters could be automatically adjusted so that the next presentation of the inputs will produce a response closer to the desired response. This learn by example method is more akin to the manner in which humans develop cognitive abilities.

Artificial neural networks have recently come into the research spotlight again. After a long sabbatical from intensive study from the mid-1960s to the mid-1980s, much progress has been made in understanding these complicated and dynamic, yet crude, models of the nervous system. It is believed by many investigators [5-7] that these models hold promise for performing many tasks for which expert systems are inadequate, inappropriate, or ineffective. These include robust image, speech, and hand-written character recognition. These tasks have eluded the grasp of symbolic processing paradigms, but seem well-suited for the learning mechanisms inherent in neural networks.

During the past year and a half, research has been conducted at the Center for Nondestructive Evaluation (NDE), Iowa State University, to investigate the learning mechanisms and network structures associated with artificial neural networks. The research was inspired by many needs within the Center for NDE for solutions to difficult inversion problems and robust signal processing methods. The application of neural networks to these and other problems has proven effective and inspiring [8-10].

This thesis covers several issues concerning the development of artificial neural networks, as well as the application of such networks to one problem identified at the Center for NDE. Chapter 2 provides a general background and brief history of the development of artificial neural networks. Chapter 3 describes in detail the mechanisms involved in a well-known learning algorithm, backpropagation. To show the merits associated with these networks, the application of this "standard" neural network paradigm to an inversion problem in NDE is described in Chapter 4. In Chapter 5, a novel learning mechanism based on genetic algorithms is described. Chapter 6 provides a comparison of the backpropagation and genetic-based learning algorithms. This comparison is made from two different points of view. The first analyzes the computational complexities of each algorithm, while the second compares the number of iterations required in obtaining a solution for several popular binary mapping problems. Chapter 7 discusses a mechanism, known as Dynamic Node Creation (DNC), first published by Timur Ash [11], applied to both backpropagation learning and genetic-based learning. This chapter includes a discussion of some relevant issues pertaining to feature detection not discussed in Ash's paper. Finally, Chapter 8 presents conclusions and a discussion of the issues presented in this thesis.

2 NEURAL NETWORKS

2.1 Overview

Artificial neural networks are intended to model the structure and operation of biological nervous systems. They are composed of simple processing elements, richly interconnected. These networks can be trained to perform arbitrary mappings between sets of input-output pairs by adjusting the weights of interconnections. They require no a priori information or built-in rules; rather, they acquire knowledge through the presentation of examples. This characteristic allows neural networks to develop mappings for functions which do not appear to have a clearly defined algorithm or theory. Further, neural network performance has proven robust when faced with incomplete, fuzzy, or novel data.

Succinctly, a neural network can be described as a directed graph, with the nodes of the graph represented by the artificial neurons (from now on referred to as neurodes--a combination of neurons and nodes), and the edges of the graph represented by the connections between the neurodes [5]. Typically, each neurode receives stimuli (input data) from several sources--other neurodes, the outside world, or both--and operates on its stimuli with a transfer function to produce a single output. This output becomes either the input to other neurodes in the network or the final output of the network. Figure 1 shows a graphical representation of a neural network with an arbitrary network architecture. In this figure the circles represent the neurodes and the lines depict the network connections.

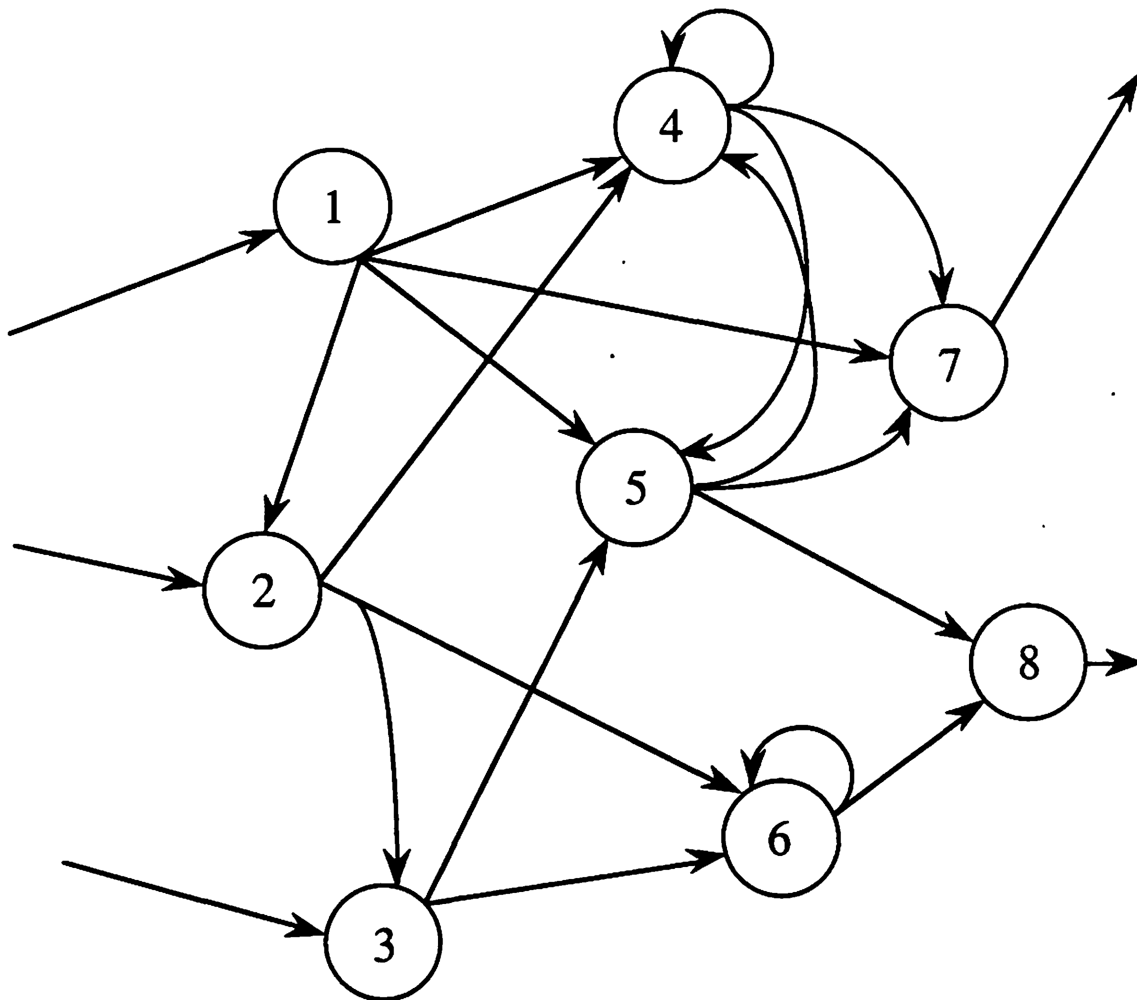


Figure 1. A graphical representation of an artificial neural network with arbitrary network architecture. The circles represent the neurodes and the lines show the network connections.

2.2 History

In order to provide motivation for some of the research presented later in this thesis, it will be helpful to provide some background and history relevant to the path which research on artificial neural networks has taken.

The first significant contribution to the idea of emulating processing functions seen in biological organisms via the implementation of simple, highly interconnected computing elements was put forth in 1943 by Warren McCulloch and Walter Pitts [12]. This work presented their ideas of combining finite-state machines and linear threshold elements for describing some forms of behavior and memory. In 1947 they published another important piece of work [13] describing network architectures with the ability to recognize spatial patterns, even when the patterns were subjected to geometrical transformations. This provided an early simulation of the processing thought to be carried out in biological visual systems.

Later that same decade Donald Hebb published a book, The Organization of Behavior, which, for the first time, attempted to describe what role networks of neurons might play in thought processing. Hebb's important contribution from this work was that networks consisting of simple neurons might learn by creating internal representations of concepts.

Not long after these works were published, research interest in this area subsided. It was not until 1962, when Frank Rosenblatt published Principles of Neurodynamics that the field of neural computing was to regain momentum. In his book, Rosenblatt proved a remarkable theorem based on a network of linear thresholding elements. He termed these networks perceptrons, shown in Fig. 2, and the theorem is called the Perceptron Convergence Theorem. It states, briefly, that a perceptron is capable of learning anything that it is possible to program it to do [2,14,15]. As significant as this result is, it also leaves perceptrons vulnerable due to limitations in their ability to handle certain types of mapping problems. For instance, because a

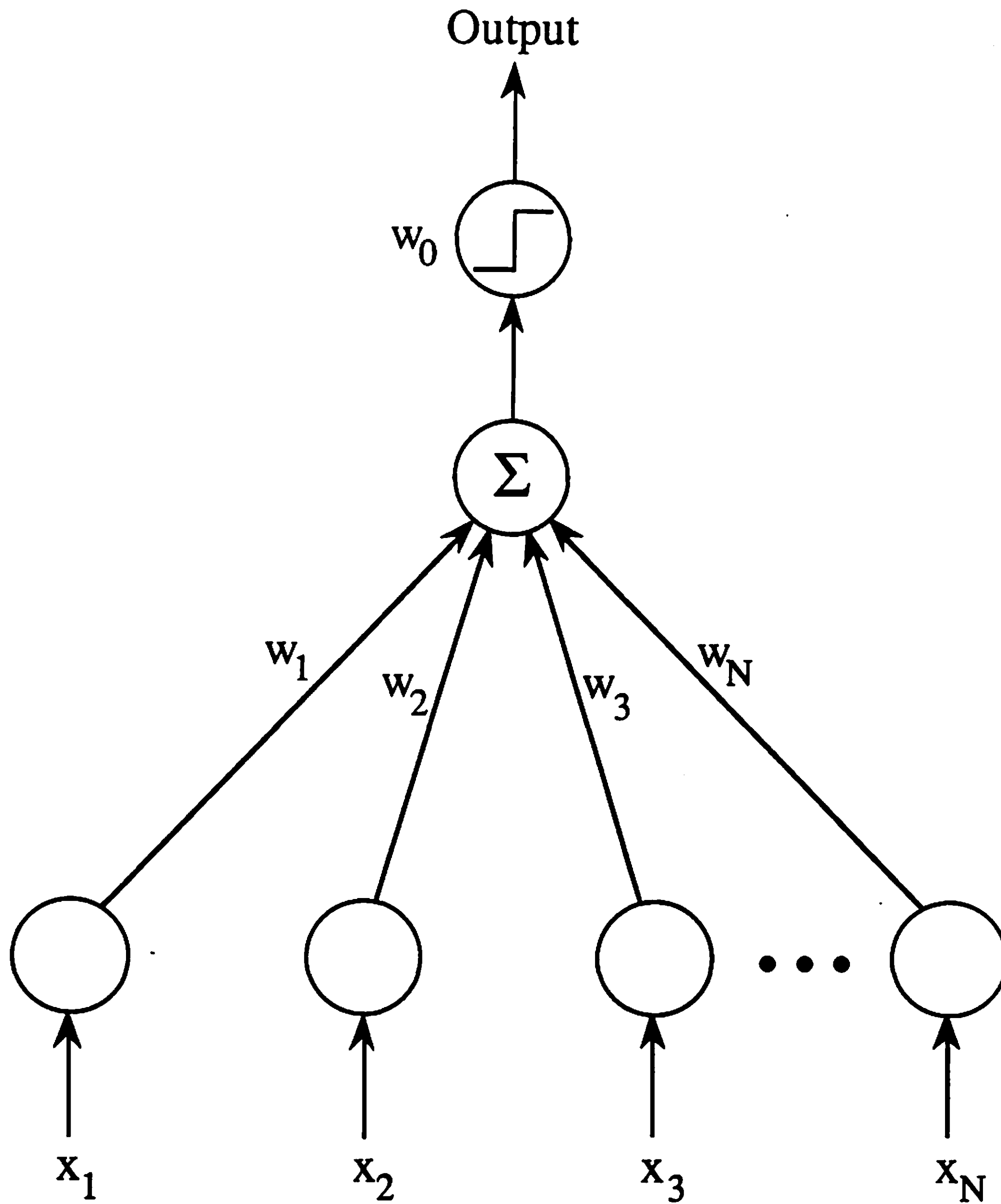


Figure 2. Graphical representation of Rosenblatt's perceptron. The output neurode (shown here as two separate circles) performs a thresholding function on the weighted sum of its inputs. w_0 is the threshold of the output neurode.

perceptron consisted of only linear thresholding elements and the inputs of a perceptron are connected directly to the outputs, Minsky and Papert [2] were able to show that there are certain simple functions for which it is impossible to program a perceptron to perform and, therefore, impossible for it to learn. The classic example of this type of function is the exclusive-OR (XOR). The reason this function can not be learned by a perceptron is that XOR is not linearly separable. Table 1 shows the input and output values for XOR, while Fig. 3 shows what is meant by not linearly separable. Any two-input function whose outputs can not be separated into distinct classes by a single straight line, or in the case of higher dimensional input vectors, by a single hyperplane, is not linearly separable. A perceptron can not learn to perform the correct mapping for any function which is not linearly separable. This work by Minsky and Papert, for all practical purposes, again halted intensive artificial neural network research for many years.

Table 1. The exclusive OR (XOR) function

Input x_1	Input x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

However, at this time it was also well-known that a perceptron consisting of a layer of hidden neurodes, neurodes that neither receive inputs from the outside world nor send outputs to the outside world, could perform the XOR function. This network is shown in Fig. 4. Unfortunately, one problem still persisted. A learning rule did not exist for adjusting the

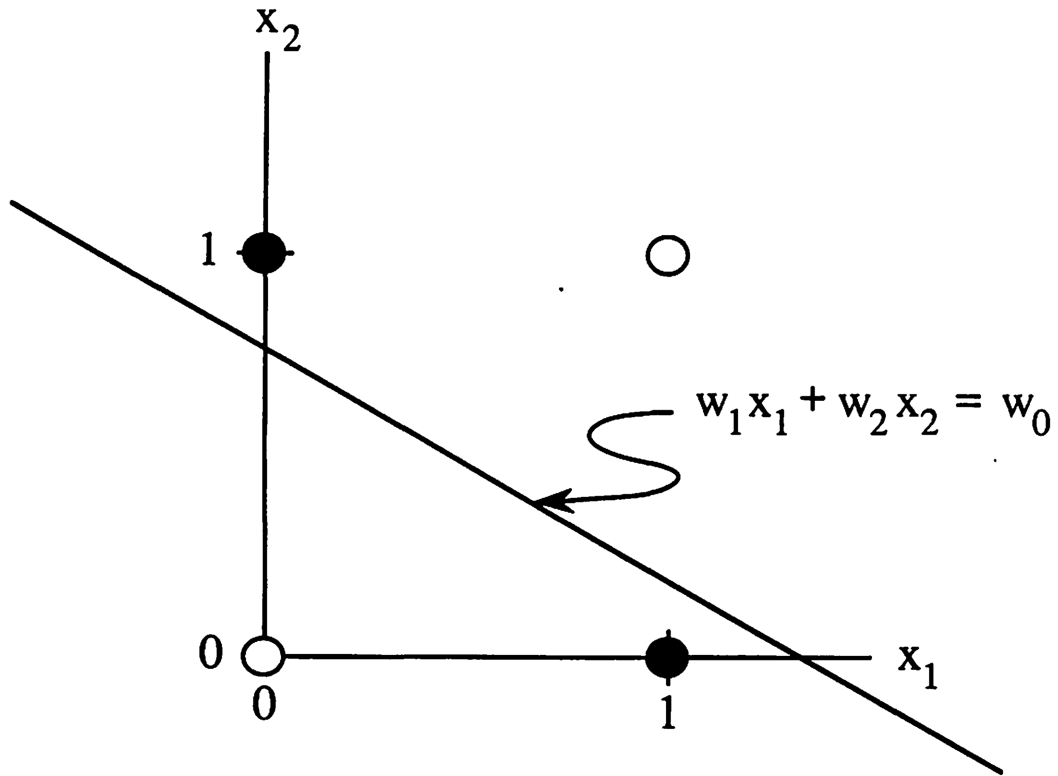


Figure 3. This graph shows a plot of the XOR function as given in Table 1. Notice that a single straight line can not be drawn which separates the output responses, denoted as a filled circle for a 1 and an open circle for a 0, into two distinct classes. This is what is meant by not linearly separable. A single layer perceptron can not solve this simple problem. The equation for the line shown in the figure is for a single layer perceptron with two inputs and a threshold w_0 on the output neurode.

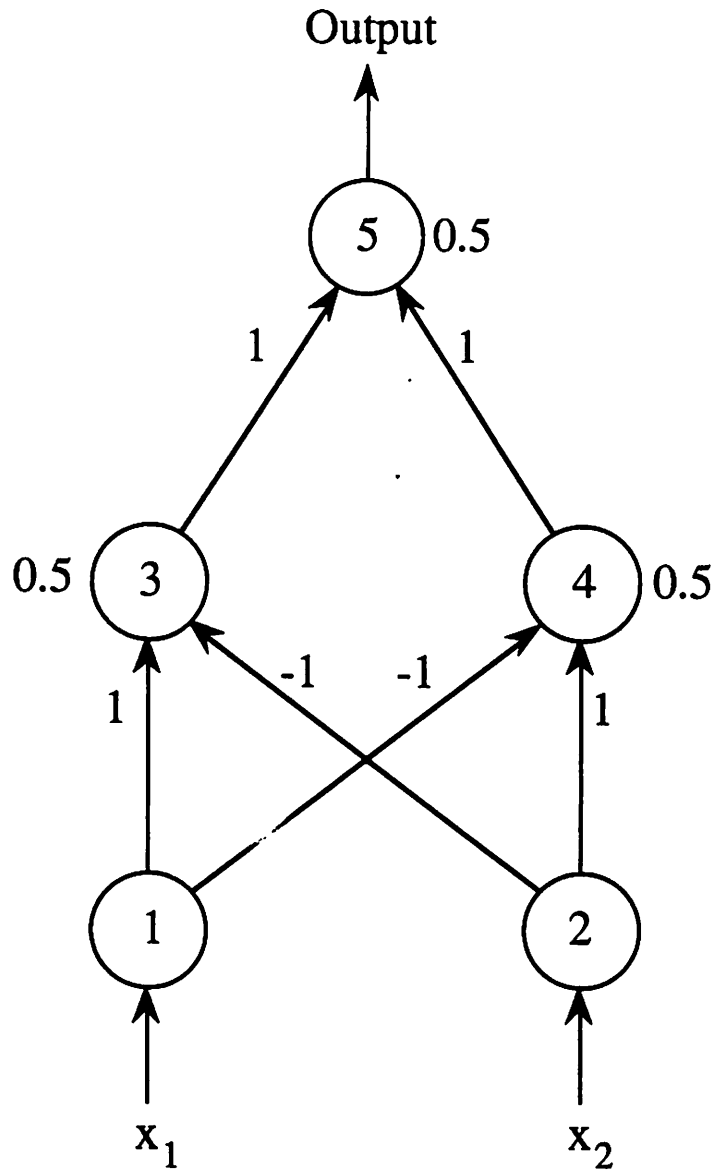


Figure 4. A multi-layer perceptron which can solve the XOR function as given in Table 1. The hidden layer is necessary (neurodes three and four) because XOR is not linearly separable. The number next to each neurode is the threshold for that neurode.

weights on the hidden layer neurodes. In other words, it was possible to program a multi-layer perceptron to perform the XOR function, but no algorithm existed by which the correct weighting factors could be learned. The Perceptron Convergence Theorem was only valid for perceptrons with an input layer and an output layer. With no hidden layer, adjusting the weights was as simple as calculating the error between the network output and the desired output and assigning an amount of this error to each of the contributing neurodes based on its input. The introduction of a hidden layer creates a credit assignment problem. That is, given that an error has occurred at the output layer, what proportion of that error can be attributed to the weights from the input layer to the hidden layer?

It was not until much later, around 1986, that this question was satisfactorily answered by Rumelhart, McClelland and the Parallel Distributed Processing (PDP) Group [16]. This work revitalized research on artificial neural networks. Their solution is known as the generalized delta rule, more commonly referred to as backpropagation, and is discussed in detail in Chapter 3. Backpropagation provides a method for adjusting weights in a multi-layer perceptron-like network, and has proven very effective. As is discussed at the end of Chapter 3, backpropagation still possesses some limitations and disadvantages.

It should be noted that Werbos [17], Parker [18], and LeCun [19] had performed work similar to that of Rumelhart et al. in 1974, 1982, and 1985, respectively. However, the work most often cited in association with backpropagation is that of Rumelhart et al. and will be used as a reference throughout this thesis.

3 BACKPROPAGATION LEARNING IN FEED-FORWARD NEURAL NETWORKS

3.1 Overview

As discussed in Chapter 2, Rosenblatt's single layer perceptron was unable to perform mappings for functions which are not linearly separable. Although a multi-layer perceptron was known to be able to perform these kinds of mappings, it was unclear what the procedure for learning the correct weighting factors on the interconnections should be. In particular, although adjusting weight connections from the hidden layer nodes to the output layer nodes was straight forward and well-understood, adjusting the weights from the input layer to the hidden layer was an enigma. This credit assignment problem stymied researchers for many years. A good solution to this problem was not widely acknowledged until David E. Rumelhart, James L. McClelland and the Parallel Distributed Processing (PDP) Group published a two volume series, Parallel Distributed Processing, in 1986. Their method for adjusting the weights in a multi-layer perceptron-like neural network is called backpropagation, due to the manner in which errors calculated at the output layer are propagated back through the network to the input layer. To date, this method of configuring the weights in a feed-forward network (Section 3.2) with supervised learning (Section 3.4) has been the most widely used and successful. The remainder of this chapter will describe in detail the method for adjusting weights in a multi-layer perceptron-like network. Section 3.2 will describe feed-forward neural networks, Section 3.3 discusses how a neurode's output is computed, and Section 3.4

presents a brief introduction to supervised learning paradigms. Next, the backpropagation learning algorithm is presented in Section 3.5, followed by an explanation of the activation function typically used in this learning paradigm in Section 3.6. Finally, in Section 3.7, the advantages and disadvantages of backpropagation are discussed in order to further establish a basis for the research presented in Chapters 5, 6 and 7.

3.2 Feed-Forward Neural Networks

Figure 1 depicted a graphical representation of a neural network with an arbitrary network architecture. This is the type of structure one might expect to find in biological nervous systems. Some neurodes, such as neurodes one, two, and three in Fig. 1 receive stimuli from the outside world. These neurodes are termed input neurodes. Other neurodes, numbers seven and eight, send their responses, or states of activation, back to the outside world and are thus called output neurodes. Neurodes four, five, and six receive stimuli only from other neurodes in the network and send their responses only to other neurodes in the network. These are called hidden neurodes. Notice that two of the neurodes in the network have recurrent connections--each sends its state of activation back to itself. Also notice that the connectivity of the neurodes in the network appears to be quite random. Although this may be true in biological organisms, this lack of structure severely complicates the analysis of artificial networks. Therefore, most researchers have opted to study networks with a more regular connectivity structure.

Typically, neurodes are grouped into layers, with neurodes in one layer connected only to neurodes in other layers, i.e. no intralayer connections and no recurrent connections. These types of networks are called feed-forward neural networks. Another restriction often applied is

limiting the reach of the output response from one layer to the next higher layer in the network. For example, in a network with an input layer, a hidden layer (a layer consisting of only hidden neurodes,) and an output layer, there could be no direct connections from the input layer to the output layer. Further, a neurode in one layer is commonly connected to all neurodes in the layer above. In this case, the network is termed a fully connected feed-forward neural network. This is the type of network which will be considered throughout this thesis. A graphical representation of a network with this structure is shown in Fig. 5.

3.3 Determining a Neurode's Output

Each neurode in a network performs a specific function. This function is to produce a response to given input stimuli. A neurode's response, or output, is computed by a transfer function. Typically, the output of a neurode is determined in two steps. The first step is to compute a weighted sum of inputs. This weighted sum of inputs is also known as the neurode's state of activation--to what degree the neurode is activated. This is done by multiplying each of the incoming signals by the weight of the connection on which the signal is received. Referring to Fig. 6, this state of activation, I, is calculated as

$$I = w_{j0} + \sum_{i=1}^{N} w_{ji} x_i, \tag{1}$$

where w_{ji} is the weight on the connection from the ith neurode in the preceding layer to the jth neurode (the neurode whose transfer function is being computed) in the current layer, x_i is the output of the ith neurode, and N is the number of input stimuli. w_{j0} is a bias term, or

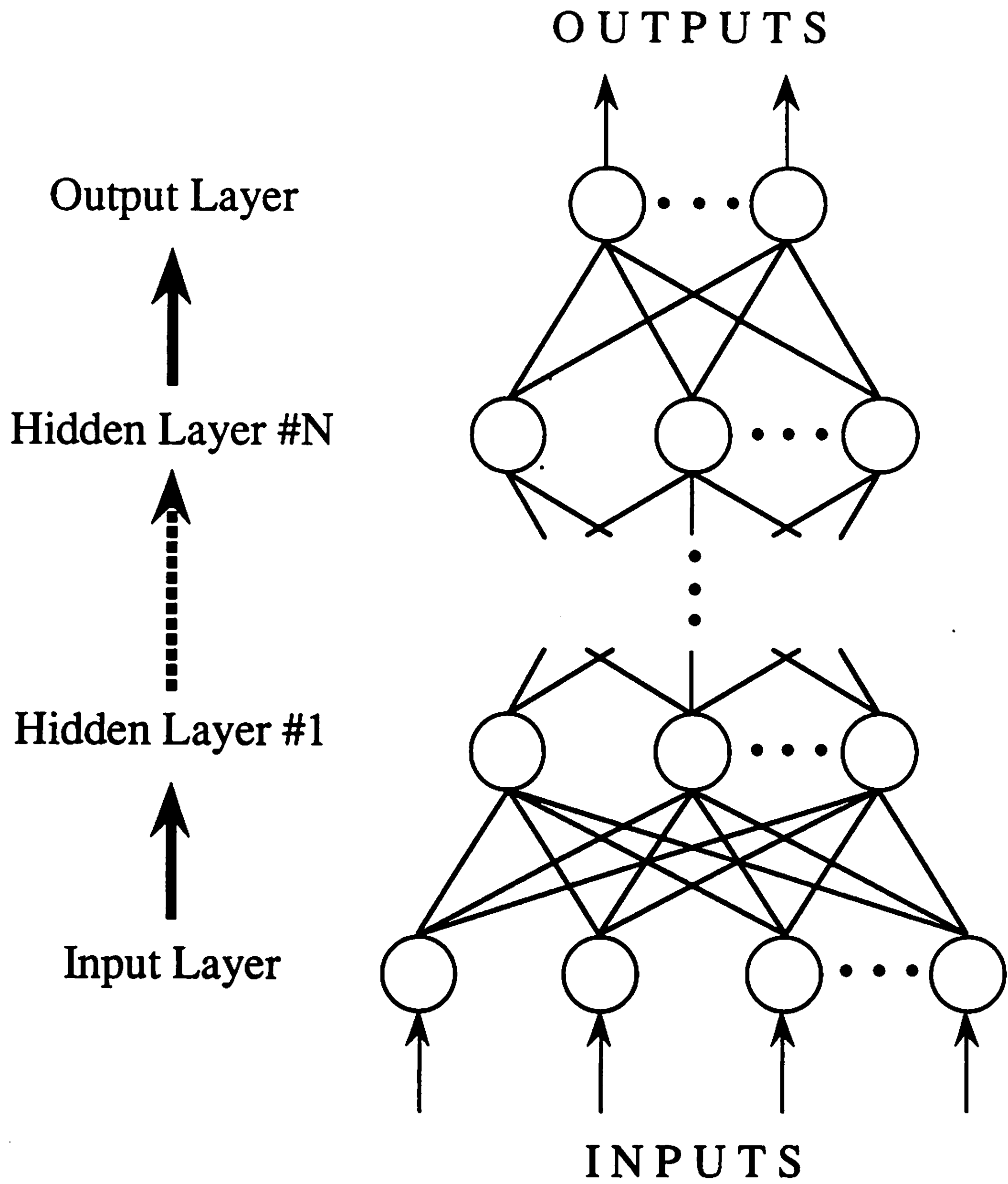


Figure 5. A fully connected, multi-layered, feed-forward neural network.

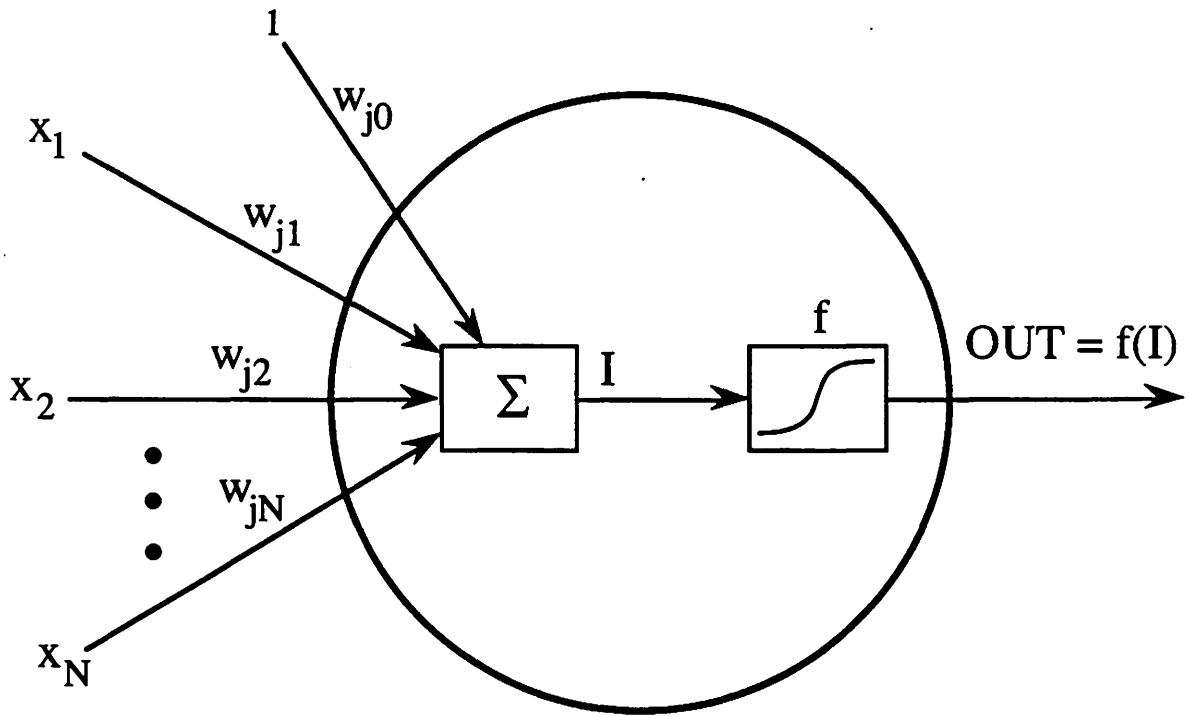


Figure 6. Abstract representation of an artificial neuron, referred to as a neurode. This neurode performs two simple steps to calculate its response based on received inputs. It first computes the weighted sum of inputs, I, and then applies an activation function, f, to I. w_{j0} is the neurode's bias term.

threshold, and will be discussed more fully in Section 3.6. The second step is to apply an activation function, f, to I in order to produce the neurode's response to the stimuli. Denoting the neurode's response as OUT,

$$OUT = f(I). \tag{2}$$

3.4 Supervised Learning

By appropriately adjusting the weights of interconnections in a neural network, the network can be configured to approximate, or learn, a particular function. Supervised learning is one of several different methods used in training neural networks. In a supervised learning environment, the correct set of output responses is known for a given set of input stimuli. This suggests that a "teacher" must be present which can guide the network in making appropriate adjustments to the network weights in order to correct errors. Weights are adaptively adjusted through the application of a learning algorithm (the "teacher") during a process called training. Training is an iterative procedure in which stimuli are presented to the input layer of the network and the network is allowed to compute its final response to this stimuli at the output layer. For a supervised learning algorithm, such as backpropagation, the outputs computed by the network are then compared against the desired outputs for the given stimuli. This comparison gives a measure of the overall error in the network, which is then used to adjust the weights in the network. After adjusting the weights, the next set of stimuli is presented, and the process continues until the error at the output layer is "acceptable" over the entire training set. The acceptable level of error for a given network is problem and implementor dependent.

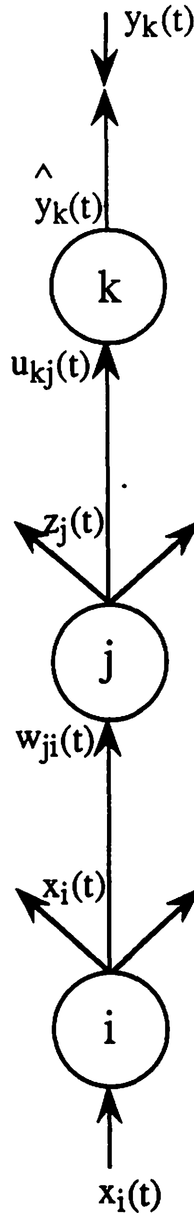
3.5 The Backpropagation Learning Algorithm

The backpropagation learning algorithm was a major breakthrough in solving the credit assignment problem in multi-layer networks. Training a neural network using the backpropagation learning algorithm consists of: 1) presenting inputs to the network's input layer; 2) allowing the network to compute its outputs; 3) computing the errors at the output layer by presenting the network with the target outputs for the associated inputs; 4) propagating error signals back through each layer in the network; and 5) adjusting all weights in the network so as to minimize the errors at the outputs. Steps one and two above represent a forward pass through the network, while steps three, four, and five represent a backward pass. Therefore, two passes through the network are required for each input-output pair--one forward to compute outputs and one backward to compute errors and adjust weights.

It will be helpful to refer to Fig. 7 which shows a portion of a multi-layer network and the relationship of the terms and symbols involved in performing a forward and a backward pass through the network. In this figure, $w_{j0}(t)$ and $u_{k0}(t)$ are the bias terms at time t for neurodes j and k, respectively. They are treated in the same manner as any other weight in the network with the exception of being connected to a neurode with a constant output of one (unity). As the forward pass is trivial, the following discussion will be concerned only with the backward pass--calculation of error signals and adjustment of weights.

To adjust the weights for neurodes in the output layer, an error signal for the tth iteration and the kth neurode, $\delta_k(t)$, is calculated as

$$\delta_k(t) = f'(I_k(t))\,(y_k(t) - \hat{y}_k(t)), \qquad (3)$$



Forward Pass

$$\hat{y}_k(t) = f(I_k(t))$$

$$I_k(t) = u_{k0}(t) + \sum_j u_{kj}(t)\, z_j(t)$$

$$z_j(t) = f(I_j(t))$$

$$I_j(t) = w_{j0}(t) + \sum_i w_{ji}(t)\, x_i(t)$$

Backward Pass

$$\delta_k(t) = f'(I_k(t))\,(y_k(t) - \hat{y}_k(t))$$

$$\Delta u_{kj}(t) = \alpha\, \delta_k(t)\, z_j(t)$$

$$u_{kj}(t+1) = u_{kj}(t) + \Delta u_{kj}(t)$$

$$\delta_j(t) = f'(I_j(t)) \sum_k \delta_k(t)\, u_{kj}(t)$$

$$\Delta w_{ji}(t) = \alpha\, \delta_j(t)\, x_i(t)$$

$$w_{ji}(t+1) = w_{ji}(t) + \Delta w_{ji}(t)$$

Figure 7. A graphical view of the processes involved in performing a single iteration of the backpropagation learning algorithm in a multi-layered feed-forward neural network.

where f' is the derivative of the activation function, $y_k(t)$ is the target output for the tth iteration, and $\hat{y}_k(t)$ is the network estimate of $y_k(t)$. Because the calculation of the error signal involves computing the derivative of the activation function, f must be differentiable everywhere. Section 3.6 contains a description and discussion of the activation function typically used for backpropagation learning. The change of weight $u_{kj}(t)$ connecting the kth neurode in the output layer and the jth neurode in the final hidden layer for the tth iteration is defined as

$$\Delta u_{kj}(t) = \alpha \, \delta_k(t) \, z_j(t), \qquad (4)$$

where α is a learning rate constant and $z_j(t)$ is the output of the jth neurode for the tth iteration. The new weight, $u_{kj}(t+1)$ is then

$$u_{kj}(t+1) = u_{kj}(t) + \Delta u_{kj}(t). \qquad (5)$$

Notice that the calculation of the error signal at the output layer is straightforward because the desired state of activation is known for each neurode in the output layer. This allows for an easily understandable adjustment of the weights connecting the final hidden layer and the output layer by moving along the gradient which will produce a lower error for the given input stimuli presented during the current iteration. This same type of weight adjustment procedure was used by Rosenblatt in his perceptron models. However, computing the error signals for neurodes at layers other than the output layer is not so straight forward. The desired response for each neurode in these layers is not provided in the training set, so the proper response for these neurodes is not explicitly known.

Accordingly, the error at the output layer must be propagated back down to previous layers in the network in order to get an idea of the desired response for each of the lower layer

neurodes. This is done by computing the contribution from a lower layer neurode to the error at the neurodes in the next higher layer. Computing the error signal for these neurodes is done by summing the next higher layer neurodes' error signals, weighted by the connection strength. This provides an estimate of how much of the error at each of the neurodes in the higher layer is due to the current neurode in the layer below. For example, the error signal $\delta_j(t)$ for the jth neurode during the tth iteration in a hidden layer is calculated as

$$\delta_j(t) = f'(I_j(t)) \sum_k \delta_k(t)\, u_{kj}(t) \tag{6}$$

and the change of weight $w_{ji}(t)$ connecting the jth neurode and the ith neurode in the previous layer for the tth iteration is defined, as before, by

$$\Delta w_{ji}(t) = \alpha\, \delta_j(t)\, x_i(t), \tag{7}$$

where $x_i(t)$ is the output of the ith neurode in the previous layer, giving the new weight $w_{ji}(t+1)$ as

$$w_{ji}(t+1) = w_{ji}(t) + \Delta w_{ji}(t). \tag{8}$$

This backward propagation of error signals continues to the first hidden layer.

It is important to note that, for standard backpropagation, the weights for a given layer in the network should not be changed before the error signals for all neurodes in the next lower layer have been computed. In other words, it is necessary for the weights to remain as they were when the response to the input stimuli was computed. The weight adjustment process is

repeated for all of the input-output pairs in the training set. Training is stopped when the errors at the output layer reach a sufficiently low level. For clarity and completeness, a flow chart of the backpropagation algorithm is presented in Fig. 8.

The goal of backpropagation learning is to perform a gradient descent search in weight space to minimize the error for all pairs in the training set. The error surface is determined by the set of possible weights for the network for a given input-output pair. As with any gradient search method, backpropagation is susceptible to becoming trapped in local minima. Several methods for escaping local minima have been proposed, including the use of a momentum term [7]. Using a momentum term essentially adds some part of past weight changes to the current weight change. This can often help move over small "bumps" in the error surface that otherwise might trap the algorithm. However, this still does not guarantee that the algorithm will find a global minimum. With the addition of a momentum term, the equation for computing the change in weights (using Eq. 4) is modified to be

$$\Delta u_{kj}(t) = \alpha\, \delta_k(t)\, z_j(t) + \eta\, \Delta u_{kj}(t-1), \tag{9}$$

where η is the momentum constant. Adding a momentum term to the standard weight adjustment term nearly always results in improved performance during training. However, proper selection of the learning constant, α, and the momentum constant, η, can be crucial factors in determining the learning time and quality of solution for a given network and application.

Another variation of backpropagation commonly used is called batching. Batching refers to how often the weights in the network are actually updated. Instead of adjusting the weights after the presentation of each input-output pair in the training set, it may be preferable to adjust the weights after accumulating errors and weight changes over a number of input-output pairs.

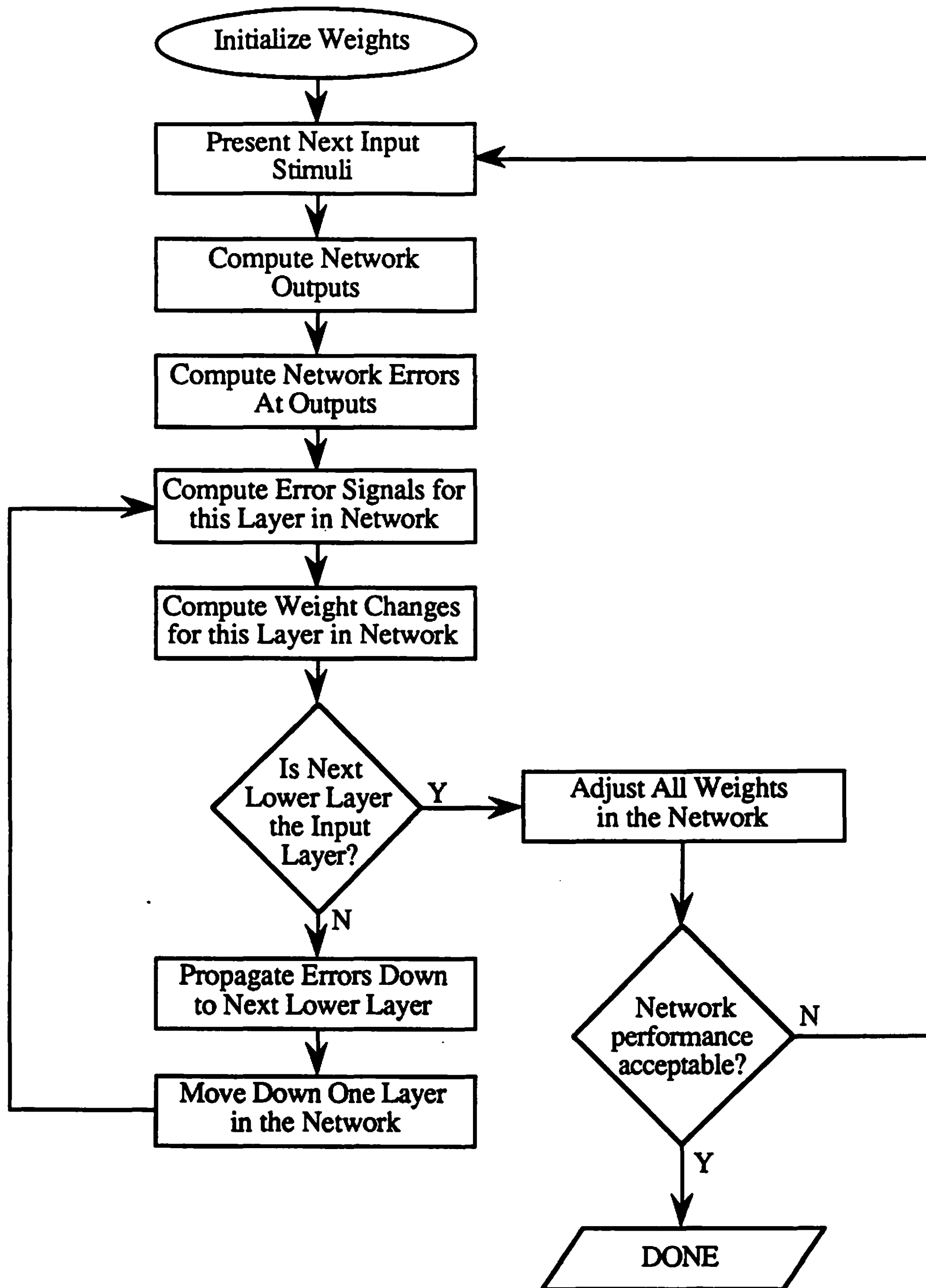


Figure 8. Flow chart illustrating the training process for backpropagation.

Batching over the entire training set more closely approaches a true gradient descent search by taking into consideration the complete function to be optimized. Adjusting the weights after the presentation of a single input-output pair can often move the weights in a direction which favors the current pair, but will have the opposite effect on the following pair or pairs. Adjusting weights after each input-output pair is equivalent to using a batch size of one.

Because backpropagation is currently the most widely used learning algorithm for multi-layered feed-forward neural networks, many other modifications to the algorithm have been proposed in order to increase performance [20-25]. Chapter 4 will present the application of backpropagation learning to a problem in nondestructive evaluation. All of the results in that chapter were obtained with the addition of a momentum term to the standard weight adjustment term.

3.6 Activation Functions

The description of the backpropagation learning algorithm in Section 3.5 showed the necessity for an activation function, f, which is differentiable everywhere. Recall that the activation function is applied to a neurode's state of activation in order to produce the neurode's output. Rosenblatt's perceptron utilized a thresholding function,

$$f(I) = \begin{cases} U & , I \geq w_0 \\ L & , I < w_0 \end{cases} \tag{10}$$

shown in Fig. 9. Here, U defines the upper bound (ON-state), L defines the lower bound (OFF-state), and w_0 is the threshold (often referred to as a bias term) When the input, I, to the function, f, equals or exceeds the threshold, the output of the function is U. Otherwise, it remains at L. w_0 has the effect of shifting the activation function along the I axis. It is clear that a derivative for this activation function does not exist everywhere, and, therefore, cannot be used for backpropagation.

Rumelhart et al. have identified the logistic (sigmoidal) activation function as one which meets several criteria deemed important in learning with backpropagation [16]. The logistic activation function, shown in Fig. 10, is given by

$$OUT = f(I) = \frac{U - L}{1 + e^{-(I * S)}} + L. \tag{11}$$

I is defined as in Eq. 1. Again, U is the upper bound, L is the lower bound, and S controls the slope of the function. Higher values of S produce steeper slopes, approaching a thresholding function, whereas lower values of S produce a more gently varying function [7]. The derivative of this activation function can be reduced to

$$f'(I) = \frac{S}{U - L} (U - OUT) (OUT - L) \tag{12}$$

where OUT is the output response of a neurode as given in Eq. 11. For an upper bound of one (U=1), a lower bound of zero (L=0), and a slope of one (S=1), Eq. 12 reduces to

$$f'(I) = (1 - OUT)\, OUT. \tag{13}$$

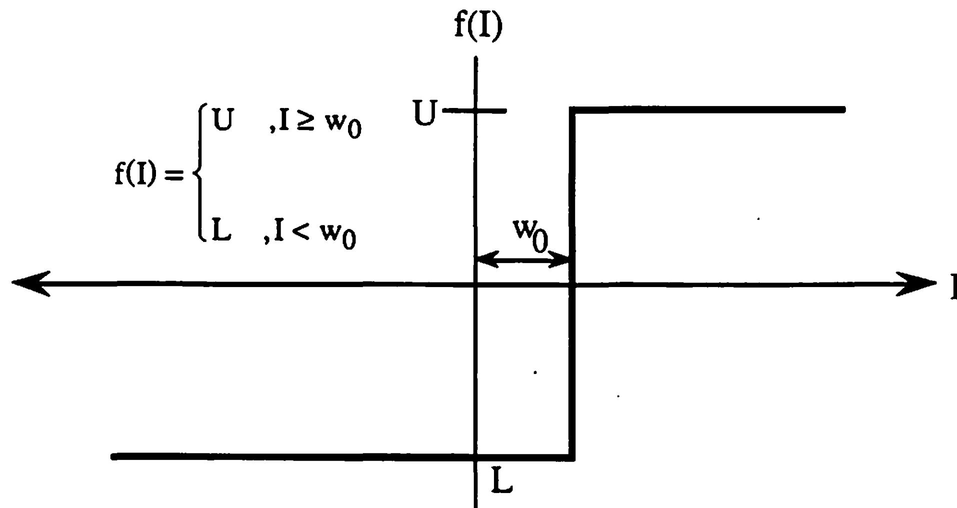


Figure 9. A thresholding activation function. U is the upper bound, L is the lower bound, and w_0 is a bias term which shifts the function along the I axis.

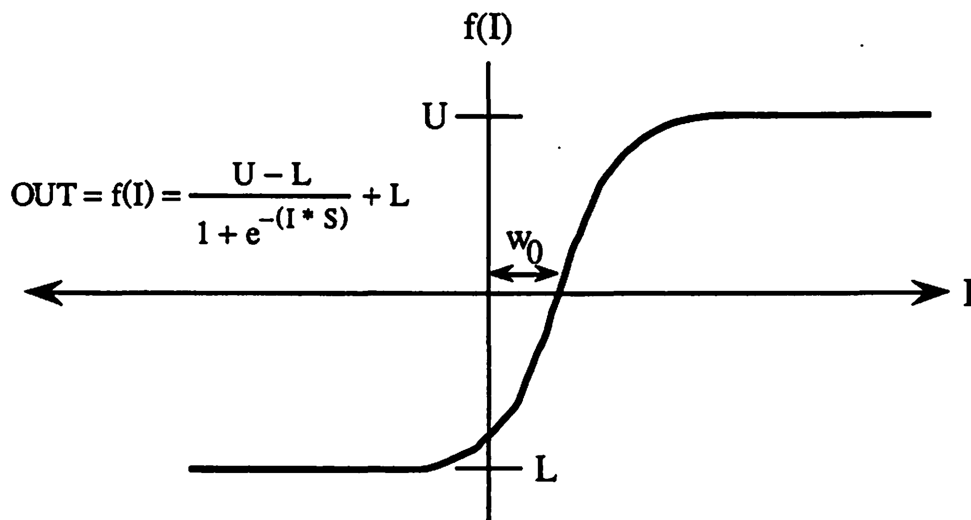


Figure 10. A logistic (sigmoidal) activation function. U is the upper bound, L is the lower bound, S controls the slope, and w_0 is a bias term which shifts the function along the I axis.

These values of U, L, and S are typical for learning binary mappings. There are several important things to notice about the logistic activation function. First, it is monotonically increasing, so that the stronger the input received by a neurode, the larger the output response will be, bounded by U. Therefore, stronger input stimuli can never produce a weaker response than weaker input stimuli. However, it also acts as an automatic gain control [6], so that as $I \rightarrow -\infty$, $OUT \rightarrow L$, and as $I \rightarrow +\infty$, $OUT \rightarrow U$. This prevents the outputs of the neurodes from growing without bound, which could allow some neurodes to dominate the operation of the network and cause a kind of "paralysis." Second, the derivative is easily computed, making it very attractive in a paradigm which can be computationally intensive. Third, the derivative attains a maximum value when OUT equals (U - L) / 2 and tails off to zero as $OUT \rightarrow U$ or $OUT \rightarrow L$. Because the amount of change for any given weight is proportional to the derivative of f, when neurodes are near their midrange of possible output responses, (U - L) / 2, the change in weights will be greatest. This suggests that until neurodes are committed to being at either their maximum or minimum output values for given input stimuli, the weights will change by a greater amount. Rumelhart et al. [16] suggest that this may contribute to the stability of learning in these networks.

3.7 Advantages and Disadvantages of Backpropagation Learning

Perhaps the greatest advantage of backpropagation is that a number of successful applications have been produced using this learning algorithm [8]. Aside from this there are several other advantages worth mentioning. First, in this era of ever-increasing computing power and speed, this algorithm performs quite efficiently and quickly on modestly sized problems. Second, because the algorithm is rather simple and recursive in nature, it is easily

implemented in software. Third, any activation function which is differentiable everywhere can be used for computing a neurode's response. Fourth, the concept of performing a gradient descent search to find minima has a long history.

However, there are a number of disadvantages associated with backpropagation, several of which are listed as advantages. First, although modestly sized problems can be solved in a reasonable amount of time, scaling problems up in size and difficulty can produce training times which run into days and weeks on even very fast computers with no guarantee of finding a "good" solution for a given network. Second, the choice of the activation function is limited to those which are differentiable. Although the logistic activation function has been useful thus far, there may exist other activation functions which are more useful or better suited to a particular application. These other activation functions may not have a derivative everywhere, or it may be difficult or expensive to calculate its derivative. Third, because backpropagation is a gradient descent technique, it is susceptible to all of the problems inherent with this technique, most notably becoming trapped in local minima. Fourth, altering the structure of the network to include recurrent connections and intralayer connections makes developing a general robust program much more complicated. Clearly some assumptions would have to be made about the order of calculation of hidden neurodes and about the recurrent connections. Rewriting the backpropagation algorithm so that it is applicable to every possible network configuration would be difficult and could lead to serious performance degradation. Obviously there are some types of networks and activation functions for which backpropagation is not well suited. Fifth, backpropagation is not well-suited for decreasing learning time through parallelization. It is a strictly iterative training procedure which is best performed on a single CPU. Certainly the neurodes in a network can be physically located on separate CPUs, but the overhead involved in communication and synchronization generally outweighs, or nearly so, the computational speed advantage of using multiple processors.

Chapter 5 presents research results on a learning mechanism which does not have any of the limitations concerning network architecture and neurode processing mentioned above. Perhaps most importantly, though, this learning algorithm is easily and naturally parallelized, significantly reducing training times on machines which are massively parallel.

4 INVERSION OF UNIFORM FIELD EDDY CURRENT DATA BY USING NEURAL NETWORKS

4.1 Overview

Nondestructive evaluation (NDE) is the discipline charged with evaluating the structural integrity and quality of materials without destroying them in the process. Over the years, many methods have been developed to interrogate materials, including visual inspection, ultrasonic, radiographic, magnetic, eddy current, penetrants, and thermographic techniques [26]. As the reliability and sophistication of these methods has progressed, NDE engineers and scientists have moved from a need for qualitative NDE (indicating the presence or absence of a flaw) to a need for quantitative NDE (determining the size and orientation of a flaw if one is present). As public demand for quality components and systems increases, quantitative NDE will surely play an increasingly important role.

During the past year and a half, research has been conducted at the Center for Nondestructive Evaluation, Iowa State University, for determining the feasibility of using artificial neural networks for interpreting signals obtained from an eddy current probe. In particular, I have investigated the use of these networks for determining the size of surface breaking cracks from data obtained with a uniform field eddy current probe.

The remainder of this chapter will discuss the type of data obtained with a uniform field probe, the steps taken for investigating the feasibility of using neural networks for interpreting these data, and the results of this investigation.

4.2 Introduction

Inversion of eddy current flaw signals has typically been based upon models of the field-flaw interaction, a so-called model-based inversion procedure. Although the feasibility of inverting eddy current data in this fashion has been demonstrated before [27-29], the complexity of such procedures has hampered their widespread acceptance and use in industry. The goal of this study is to develop an inversion method that is easy to use and implement outside the research community. This chapter presents results of the use of neural networks for the inversion of eddy current flaw signals to obtain flaw sizes.

4.3 Uniform Field Eddy Current Probe

A uniform field eddy current probe was selected for use in this study because a substantial body of experimental and theoretical work exists. The theory for the interaction of a spatially uniform electromagnetic field with a three-dimensional flaw developed by B. A. Auld et al. [30,31] is well-known and has been shown to agree with experiment [32,33]. In the limit of small skin depth, the change in probe impedance caused by a flaw can be represented as

$$\Delta Z = \frac{c}{\sigma}\frac{H^2}{I^2}\left[\Sigma^0 + (1+i)\frac{c}{\sigma}\Sigma^1 + \frac{i\,\Delta u\, c}{\sigma^2}\Sigma^1\right], \qquad (14)$$

where $\frac{H}{I}$ is the magnetic field strength per unit current, σ is the conductivity of the material under interrogation, 2c is the flaw length, Δu is the flaw width, δ is the electromagnetic skin

depth, and i represents the imaginary number, $\sqrt{-1}$. Σ^0 and Σ^1 are shape factors that depend only on $(\frac{a}{c})$, where a is the flaw depth. The three terms in Eq. 14 correspond roughly to resistive losses at the crack corners, the wall impedance of current flowing over the flaw surfaces, and Faraday induction due to the volume enclosed by currents encircling the flaw [27].

The Σs for use with rectangular and semi-elliptical flaws have been calculated numerically and fit to polynomials by a nonlinear least-squares method [34], giving

$$\Sigma^0 = -3.00 - 1.35\left(\frac{a}{c}\right) + 0.66\left(\frac{a}{c}\right)^2 \tag{15}$$

and

$$\Sigma^1 = -0.02 + 2.56\left(\frac{a}{c}\right) + 1.11\left(\frac{a}{c}\right)^2 - 1.70\left(\frac{a}{c}\right)^3. \tag{16}$$

The probe used here is based on a design proposed by E. Smith [32], but in a slightly different configuration [34], as shown in Fig. 11. This probe consists of a C-shaped ferrite core wound with 65 turns of wire, creating an active region between the two tips of the ferrite. To increase uniformity of the magnetic field in the active region, the tips of the ferrite were shaped and chamfered [34], and a copper foil surrounds the probe to diminish effects of field leakage.

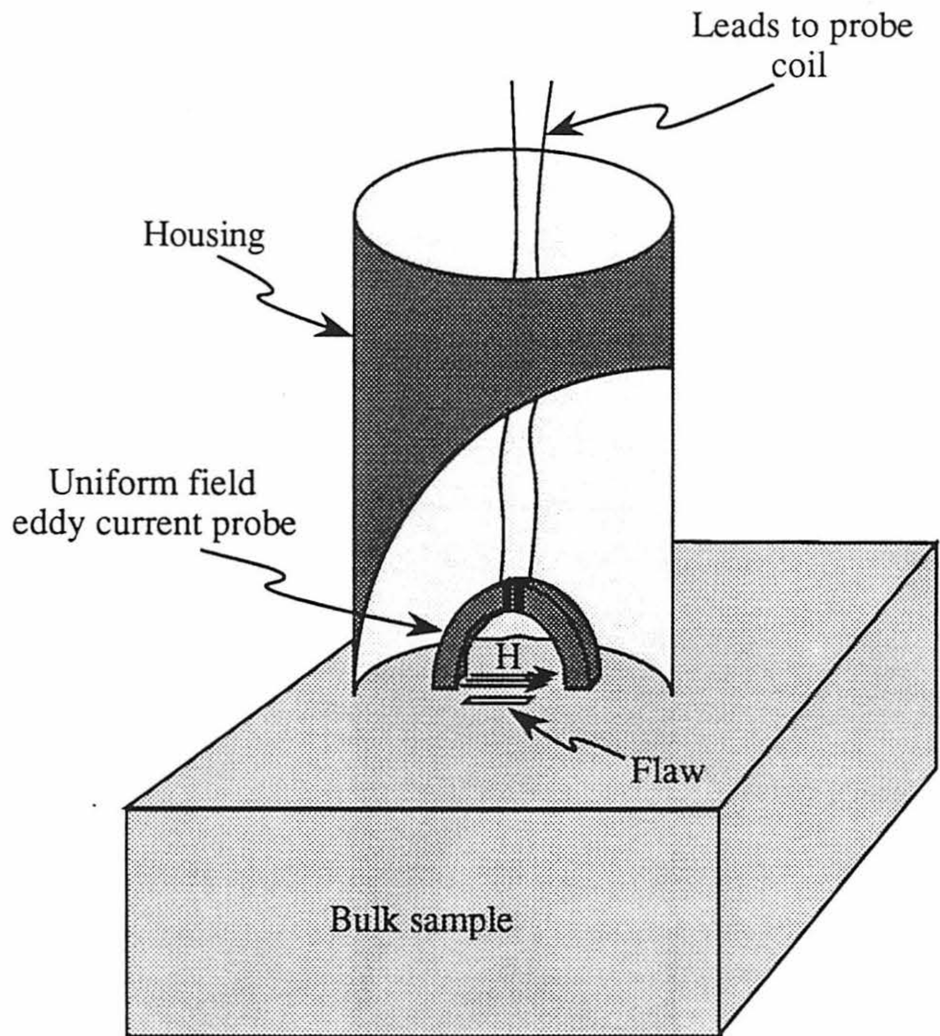


Figure 11. Uniform field eddy current (UFEC) probe.

4.4 Approach

Two approaches were taken to determine the feasibility of using a neural network for inversion of eddy current flaw signals. The first approach involved the development of a network for inverting synthetic (noise-free) data. Flaw dimensions were generated randomly such that $c \leq 2.00$ mm, $\frac{a}{c} \leq 1$, $\Delta u \leq 0.1c$, and $\frac{a}{\delta} \geq 2$, where c is the flaw half-length, a is the flaw depth, Δu is the flaw width, and δ is the skin depth. These limits were chosen to assure compatibility of synthetic data with experimentally accessible flaw dimensions.

The second approach was the trial inversion of experimental data obtained with a uniform field probe. Data were taken with the uniform field probe described earlier using computer controlled x-y positioners to move a sample under the stationary probe. Real and imaginary values of probe impedance were acquired by a personal computer connected to a Hewlett-Packard 4194A Impedance Analyzer over an IEEE-488 bus. Each measurement consisted of scanning the probe over a flaw in one-dimension, giving impedance values at a number of points both near the flaw and away from the flaw. This allowed for preprocessing of the data to remove effects caused by tilt. Measurements were taken at ten frequencies between 1 and 10 MHz, spaced at 1 MHz intervals. Flaw impedance magnitude, $|\Delta Z|$, and phase, θ, at the center of the flaw were then calculated at each frequency. The flaws consisted of five semi-elliptical EDM notches, one "no flaw," and two fatigue cracks in Ti-6Al-4V. The dimensions of these flaws are shown in Table 2.

In both approaches, the flaw dimensions are the outputs of the neural network and the ΔZ magnitude-phase information is the input. The specific network architectures used for each case are discussed below. After training, the networks were tested with data not used during training in order to evaluate the network's ability to generalize.

Table 2. Dimensions and types for flaws used in the experimental portion of this study

Flaw #	Length, 2c (mm)	Depth, a (mm)	Width, Δu (mm)	Flaw Type
1	2.48	1.05	0.16	EDM notch
2	2.01	0.85	0.20	EDM notch
3	1.60	0.63	0.12	EDM notch
4	1.18	0.40	0.12	EDM notch
5	0.61	0.33	0.10	EDM notch
6	0.00	0.00	0.00	EDM notch
7	0.99	0.33	0.00	Fatigue crack
8	0.98	0.33	0.00	Fatigue crack

4.5 Implementation

When this project began, it was necessary to develop a backpropagation program that could run on a standard digital computer such as a PC, Macintosh, or Apollo workstation. The program used for this implementation was adapted from a program developed by Pao [7]. Since then, a backpropagation simulator has been developed that utilizes a coprocessor board and software library routines from Hecht-Nielsen Neurocomputers (HNC). The coprocessor board provides much greater throughput (a must for most neural network algorithms), while the software library allows for fairly easy adaptation and implementation of many different neural network paradigms.

4.6 Results Using Synthetic Data

Two sets of flaw dimensions and associated ΔZ magnitude-phase information were generated for training a network. One set consisted of 100 pairs, the other consisted of 1000 pairs. The second set included the same 100 pairs as the first set, but with an additional 900 pairs. Flaw impedance magnitude, $|\Delta Z|$, and phase, θ, were then calculated at seven frequencies (2 MHz - 8 MHz at 1 MHz intervals) according to Auld's ΔZ theory [31] for each of the flaw geometries. After training separate networks, one using the 100-pair training set and the other using the 1000-pair training set, the performance of each network was tested with a 100-pair testing set different from either of the training sets. The estimated flaw sizes computed by the network for the testing set were then compared with the actual flaw dimensions to evaluate the network's performance. The network trained on 100 pairs had an input layer of 14 processing elements (magnitude and phase for each of seven frequencies), one hidden layer of 14 processing elements, and an output layer of 3 processing elements (one each for flaw length, depth, and width). The network trained on 1000 pairs differed only by having an additional hidden layer of 14 processing elements. No attempt was made here to find minimal and/or optimal network architectures.

Figure 12 shows the results of both networks' estimates of flaw depth vs. the actual depth for each of the 100 flaws in the testing set. The figure shows that the network trained on 100 pairs (Fig. 12a) has fair performance, but that the network has not been able to closely approximate the function involved. The mean absolute error for the testing set is 16.81% and the standard deviation is 12.37%. The network trained on 1000 pairs (Fig. 12b), however, produced much better results, having a mean absolute error of 3.05% and a standard deviation of 2.89%. This demonstrates that the second network has been able to more closely approximate the mapping for flaw depth inversion.

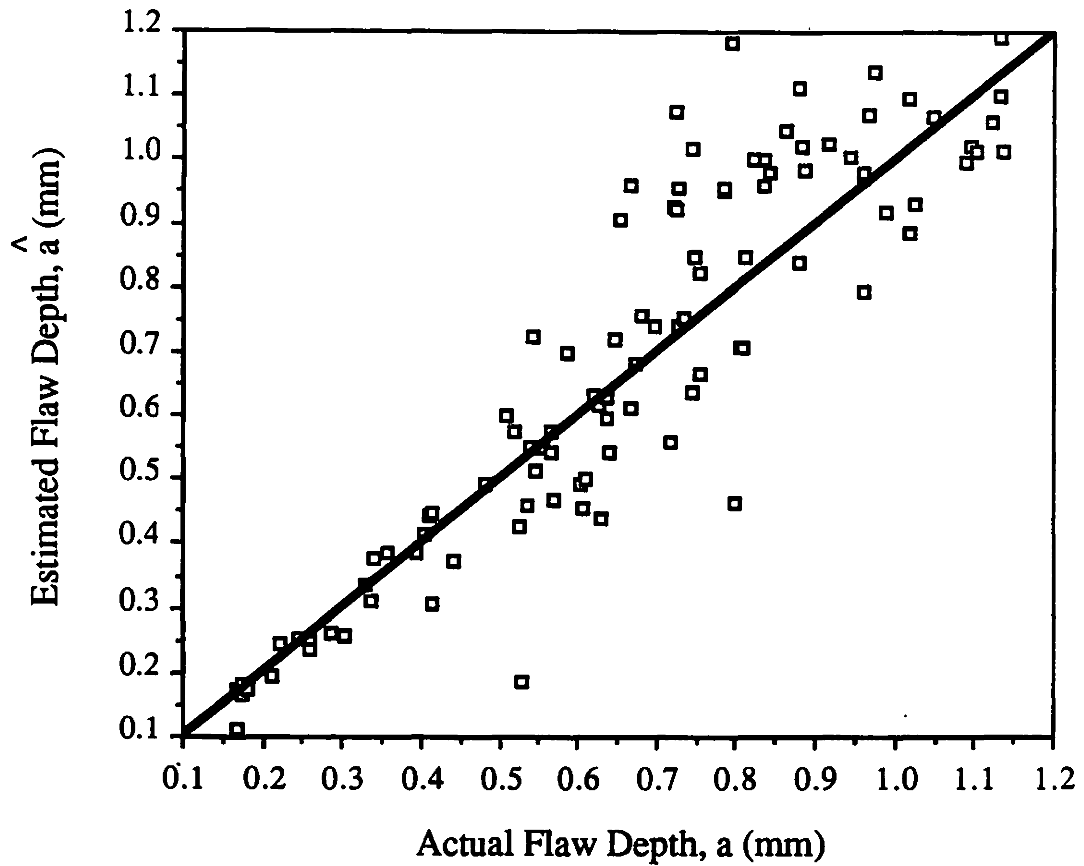


Figure 12a. Results of testing after training a neural network with synthetic data generated according to Auld's ΔZ theory for 100 randomly chosen flaw geometries. The graph shows estimated flaw depth vs. actual flaw depth.

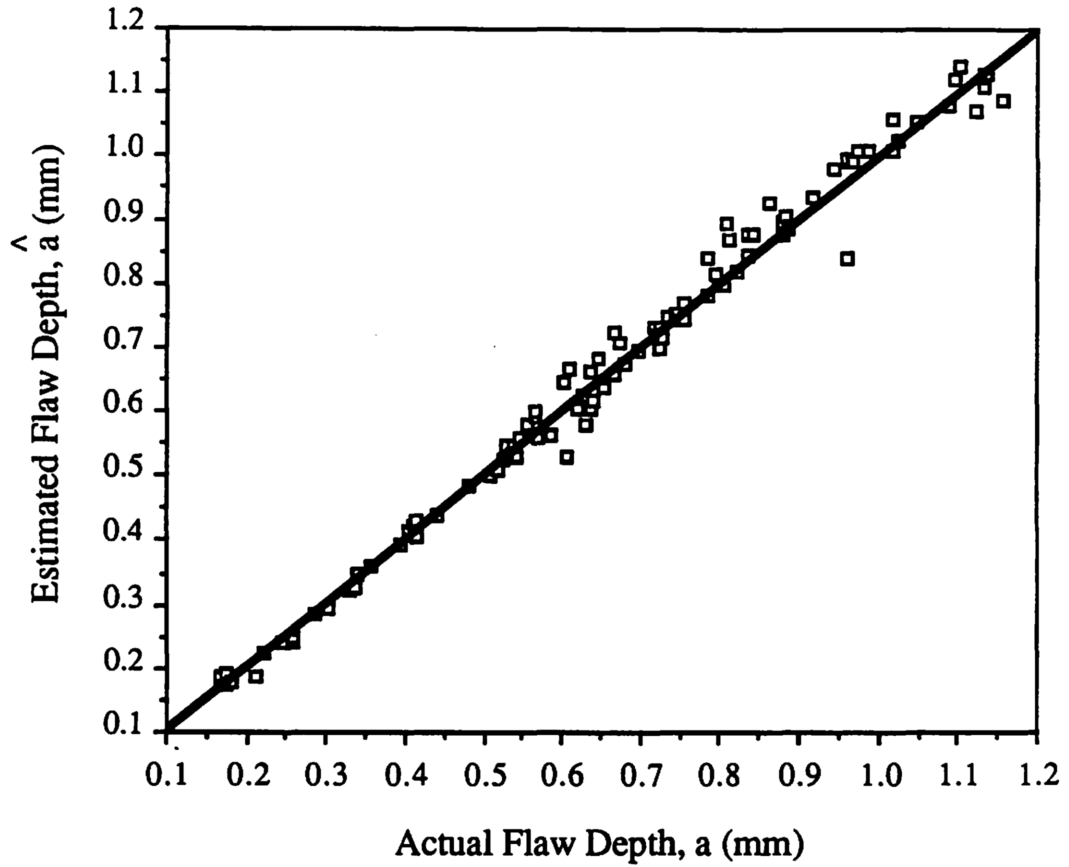


Figure 12b. Results of testing after training a neural network with synthetic data for 1000 randomly chosen flaw geometries. The graph shows estimated flaw depth vs. actual flaw depth.

Both training sets represent a small fraction of the total population of flaws. This implies that the performance of the first network is good considering the training set size, but that performance was dramatically improved by training on a larger set of flaws as demonstrated by the second network. This suggests that the performance of the network might be further improved by using a still larger training set.

4.7 Results Using Experimental Data

Owing to the limited number of flaws available for measurement, only one set of data was obtained for training a network to invert experimental flaw data. In order to generate training and testing sets of useful size, 20 independent measurements at ten frequencies (1 MHz - 10 MHz at 1 MHz intervals) were made on each of the eight flaws in Table 2, giving a total of 160 measurements.

Two different approaches were pursued for training and testing a network on the experimental data. The first was to use only the EDM notches and the "no flaw." Training and testing sets were created by dividing the six flaws into two disjoint subsets. Because the training set needed to span the range of flaw dimensions that would be seen during testing, I chose to use four flaws, #1, #3, #5, and #6, for training and the remaining two flaws, #2 and #4, for testing. Although the training flaws cover the range of dimensions of the test flaws for both the flaw half-length and depth, the width for flaw #2 falls outside those in the training set.

Figure 13a is a comparison of the network's estimates for flaw half-length for each of the 20 measurements on flaws #2 and #4. The network's estimates show good agreement with the actual flaw half-length, and all estimates are within about ±10% of the actual size. This net

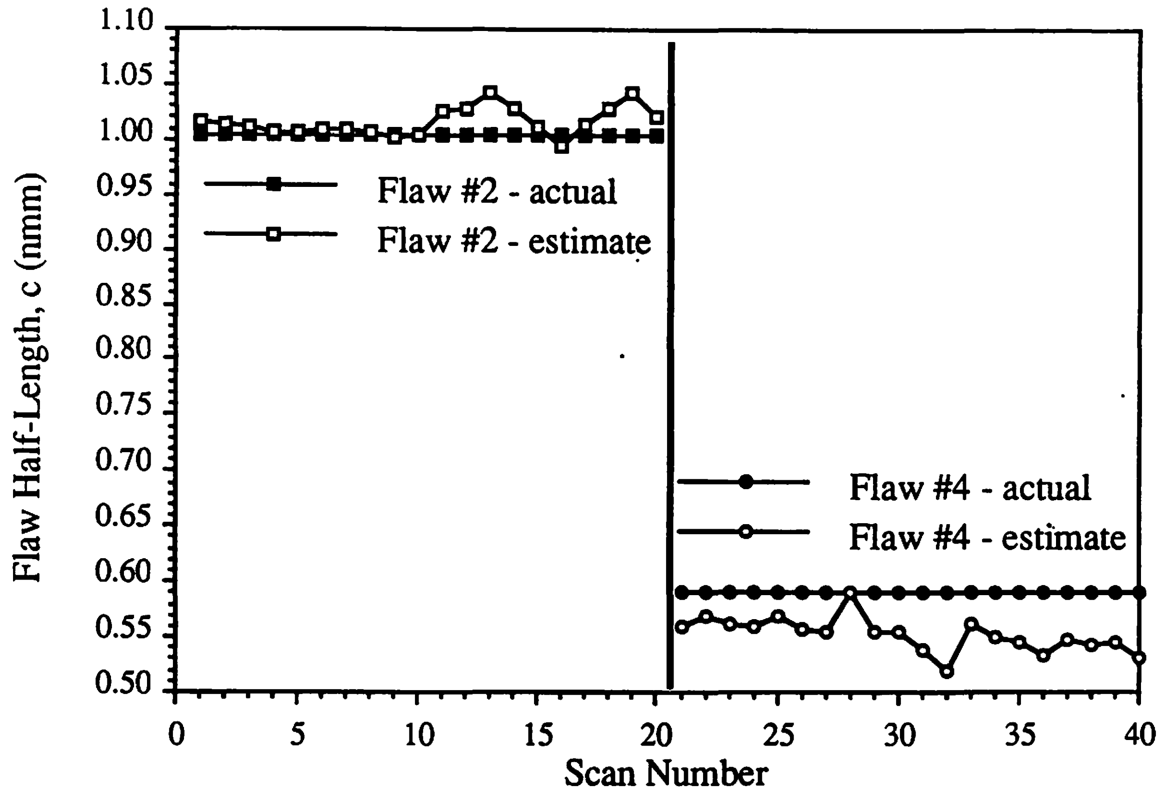


Figure 13a. Comparison of actual and estimated flaw half-lengths for flaws #2 and #4 (EDM notches). The half-lengths were estimated by a neural network after training on flaws #1, #3, #5, and #6 (see Table 2).

consisted of 20 inputs, 6 hidden neurodes, and one output and required approximately 50,000 iterations (presentations of the training set).

Figure 13b shows a comparison of the network's estimates for flaw depth with the actual depth for the same measurements on flaws #2 and #4. Again, the results show good agreement between network estimates and actual depth. The net developed for this particular inversion consisted of 20 inputs, 10 processing elements in the first hidden layer, three processing elements in the second hidden layer, and a single output (flaw depth). The number of iterations required for this net was approximately 4,000.

The next step was to see if a net could be developed to estimate flaw dimensions for a fatigue crack, having trained on only EDM notches. Thus, the five EDM notches and one "no flaw" were used in training a network, and the two fatigue cracks were used for testing. A fatigue crack differs greatly from an EDM notch in its geometrical properties. Fatigue cracks are the types of cracks to be expected when looking for flaws with an eddy current probe in an industrial setting. However, fatigue cracks of known dimension are much harder to manufacture or acquire and consequently are quite expensive and hard to come by. The goal of this particular experiment was to determine if a network could be trained using only EDM notches, yet be used to correctly size fatigue cracks. If possible, this would alleviate the burden of acquiring a large set of expensive fatigue cracks for use during training.

Figure 14a shows the comparison between network estimates for flaw half-length and actual half-length, while Fig. 14b compares estimated and actual flaw depth. It can be seen that agreement is again quite good. Further, this demonstrates the ability of the network to generalize by estimating half-length and depth for a novel flaw with novel dimensions.
Although figures showing estimates of flaw width are not included, the results of depth estimation for the theoretical case are similar to estimations for the other two dimensions. In

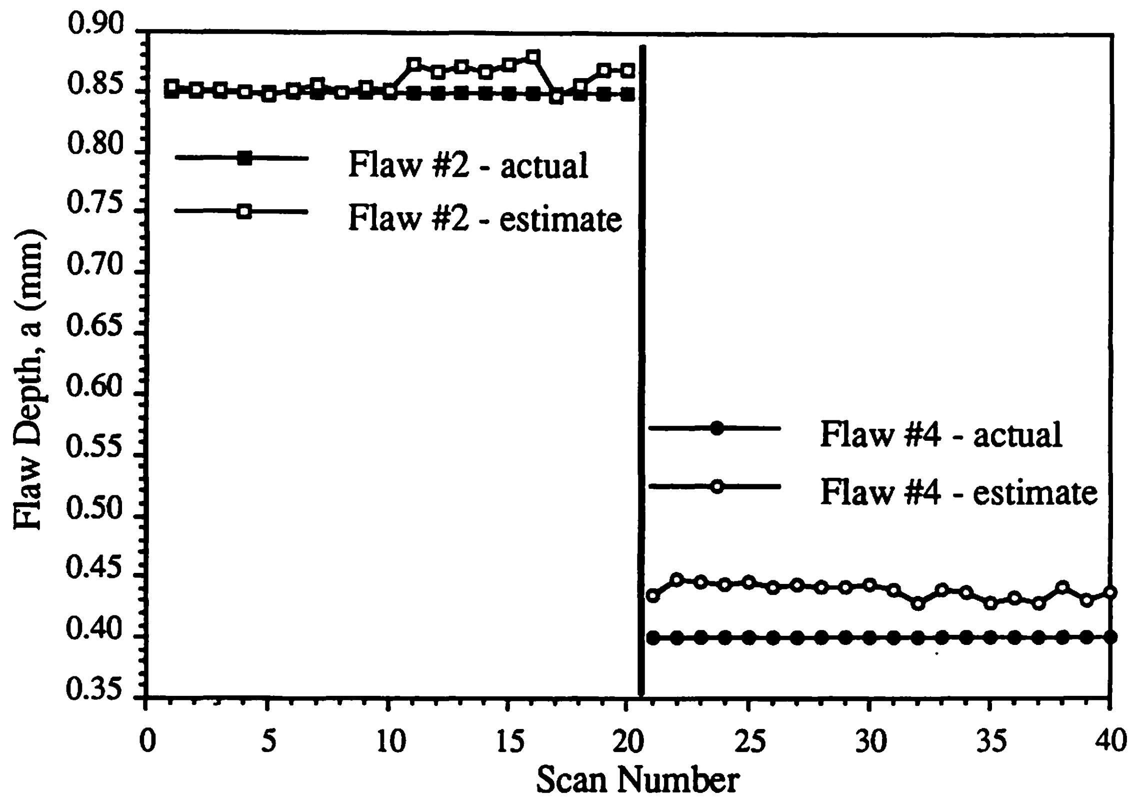


Figure 13b. Comparison of actual and estimated flaw depths for flaws #2 and #4 (EDM notches). The depths were estimated by a neural network after training on flaws #1, #3, #5, and #6.

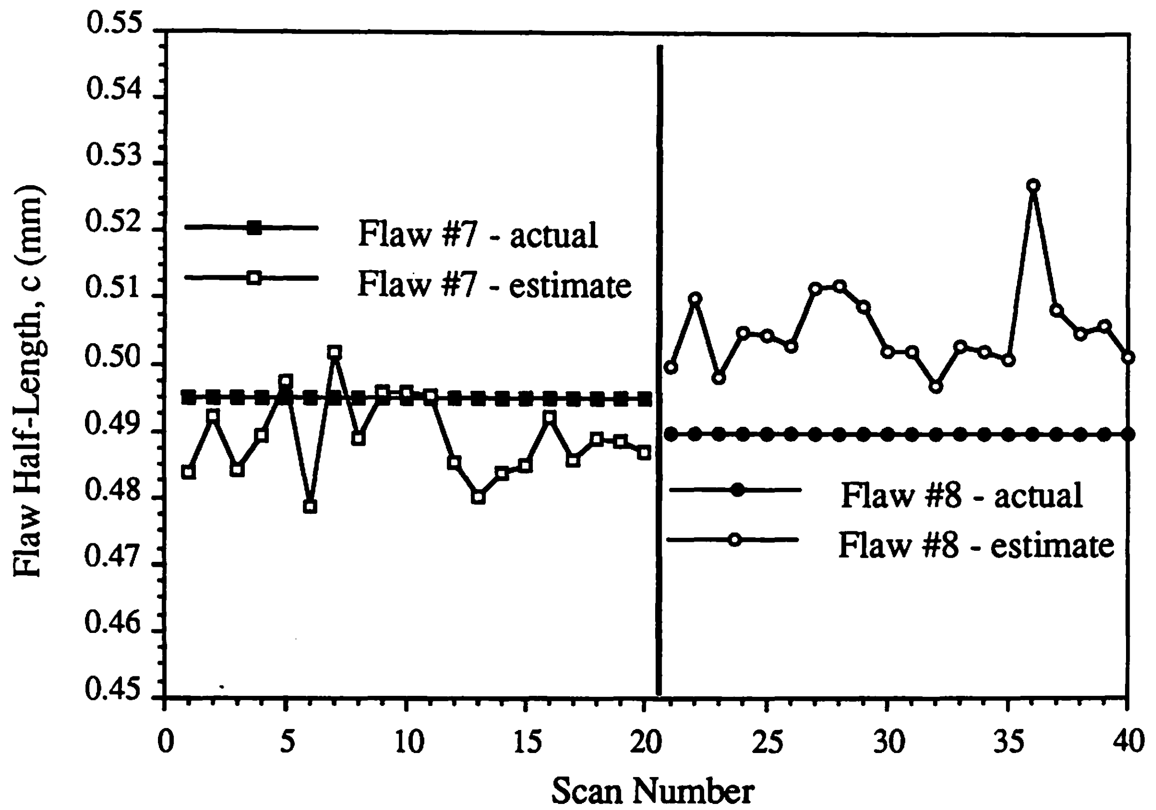


Figure 14a. Comparison of actual and estimated flaw half-lengths for flaws #7 and #8 (fatigue cracks). The half-lengths were estimated by a neural network after training on flaws #1-#6 (EDM notches).

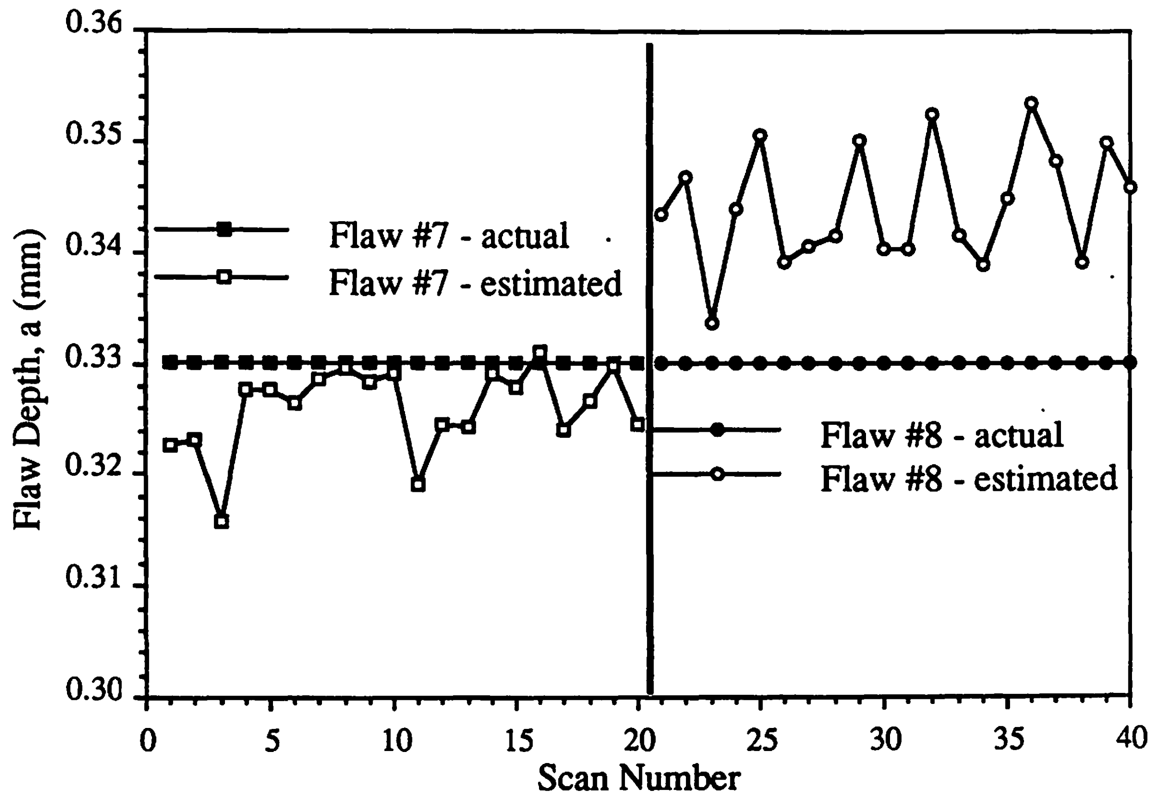


Figure 14b. Comparison of actual and estimated flaw depths for flaws #7 and #8 (fatigue cracks). The depths were estimated by a neural network after training on flaws #1-#6 (EDM notches).

the experimental case, width estimations were not attempted on the fatigue cracks since the training set could not be sufficiently configured to account for such tight flaws.

4.8 Discussion

Several steps can be taken to improve performance of the networks, particularly for experimental data. One such step is to measure more flaws, thereby creating a larger training set. I believe this will improve performance for experimental data as it did with synthetic data. This means, however, measuring a large number of flaws covering a wide range of shapes and sizes. One way to overcome this problem might be to train a network with synthetic data, from which a large training set can easily and readily be obtained, and then test the network with experimental data. This would provide a virtually unlimited training set size which could be configured to meet any criteria. The major obstacle to this approach currently rests in the inability to obtain a calibration which provides good agreement between theoretical calculations and experimental measurements over a wide range of frequencies.

Only a limited amount of effort was devoted to optimizing these networks. Two such efforts were neurode addition [11] and neurode pruning [35]. These two ideas can be combined to help find an "optimal" network configuration. For example, an initial guess at the size of the network is made. As learning progresses, it may be discovered that the current configuration will reach a point at which learning will cease (or nearly so). At this point, it may be prudent to add another processing element to a hidden layer and continue training. The addition of the extra neurode can give the network additional degrees of freedom and aid in learning. This process continues until the net is sufficiently trained or it is deemed that an entirely new structure or paradigm is required. This idea, including my variation, will be

covered fully in Chapter 7. Neurode pruning is just the opposite. After training is finished (or nearly so), it might be noted that one or more of the processing elements in a hidden layer is contributing very little to the final result. In this case, the neurode may be pruned from the network. The net should then be briefly retrained in order to account for the lost neurode. After performing several experiments, I determined that neurode addition is a more feasible approach, thus neurode pruning was dropped.

Again, only a limited amount of time was invested in exploring these optimization ideas for this application. While backpropagation is the most popular learning paradigm currently in use and generally works quite well, it has several drawbacks. One of these is that training can often be quite long, both in number of iterations required and in real processing time. For example, the net trained on flaws #1, #3, #5, and #6 required approximately 50,000 iterations and it is believed that this network is still not fully trained. It was stopped due to time considerations. Another problem is backpropagation's susceptibility to local minima as discussed earlier. This can put the net into a non-optimal state, in which case training may have to be restarted using different initial weight values. However, whether or not the net is in an optimal state may not be readily apparent.

4.9 Conclusions

Results from both the theoretical and experimental approaches to training neural networks for the inversion of uniform field eddy current data are very encouraging. There is certainly much more work to be done, however, especially with experimental data. Results from both parts of this study have shown the need for proper training sets: a large number of examples as demonstrated by the theoretical results, and a thorough and complete coverage of ranges as

demonstrated by the experimental results. As the results demonstrate, neural networks show great promise in being able to solve the inverse problem for eddy current data.

5 GENETIC-BASED LEARNING IN ARTIFICIAL NEURAL NETWORKS

5.1 Genetic Algorithms

A novel learning mechanism for artificial neural networks, genetic-based learning, is presented in this chapter. Section 5.1 is a brief introduction to genetic algorithms, their mechanisms for global optimization, and an analysis of the way in which genetic algorithms are able to search for highly-fit individuals from an initially random population of possibly poorly-fit individuals. Section 5.2 presents the application of genetic algorithms to the task of optimizing weights in neural networks. Section 5.3 discusses several improvements to the basic algorithm developed in Section 5.2.4, perhaps, most importantly, a parallel implementation of genetic-based learning presented in Section 5.3.4, which provides a significant reduction in learning time.

5.1.1 Introduction

Genetic algorithms (GAs) are a global optimization technique based on the operations observed in natural selection and genetics [36]. They operate on string structures, typically a concatenated list of binary digits representing a coding of the parameters for a given problem. Many such string structures are considered simultaneously, with the most fit of these structures

receiving exponentially increasing opportunities to pass on genetically important material to successive generations of string structures. In this way, GAs search from many points in the search space at once and yet continually narrow the focus of the search to the areas of the observed best performance.

Genetic algorithms differ from more traditional optimization techniques in four important ways [36]:

1.) GAs use objective function information (evaluation of a given function using the parameters encoded in the string structure) to guide the search, not derivatives or other auxiliary information;
2.) GAs use a coding of the parameters used to calculate the objective function in guiding the search, not the parameters themselves;
3.) GAs search from many points in the solution space at one time, not a single point;
4.) GAs use probabilistic rules, not deterministic rules, in moving from one set of solutions (a population) to the next.

5.1.2 Genetic algorithm terminology

To capture the spirit of genetic algorithms, it will be helpful to discuss the language of genetics and natural selection as used here. The string structures discussed above are equivalent to genotypes in genetics. Genotypes are composed of one or more chromosomes; chromosomes are referred to as strings in GAs. The organism formed by the interaction of a genotype with its environment is called a phenotype, while in artificial systems, the string structures are decoded to obtain a parameter set. A population is composed of many

genotypes. In genetics, chromosomes are composed of genes, which may take on some number of values, called alleles. In genetic algorithms, genes are referred to as features. Features may be located at different positions in a string (chromosome). In genetics, this position is called the gene's locus.

Table 3 lists a comparison of the terms used in natural genetics and in genetic algorithms, while Fig. 15 shows the relationship of these terms in the context of a sample genotype. If there is only one chromosome in a genotype, then the string (chromosome) and its structure (genotype) could be referred to interchangeably.

Table 3. Comparison of natural genetics and genetic algorithm terminology

Genetic Algorithms	Natural Genetics	Description
Structure	Genotype	Total genetic package
Strings	Chromosomes	One or more combine to form a total genetic package
Features	Genes	What chromosomes are composed of
Values	Alleles	Values which genes may have
Position	Locus	Position of a gene in a chromosome
Parameter Set	Phenotype	Organism formed by interaction of the genotype and its environment

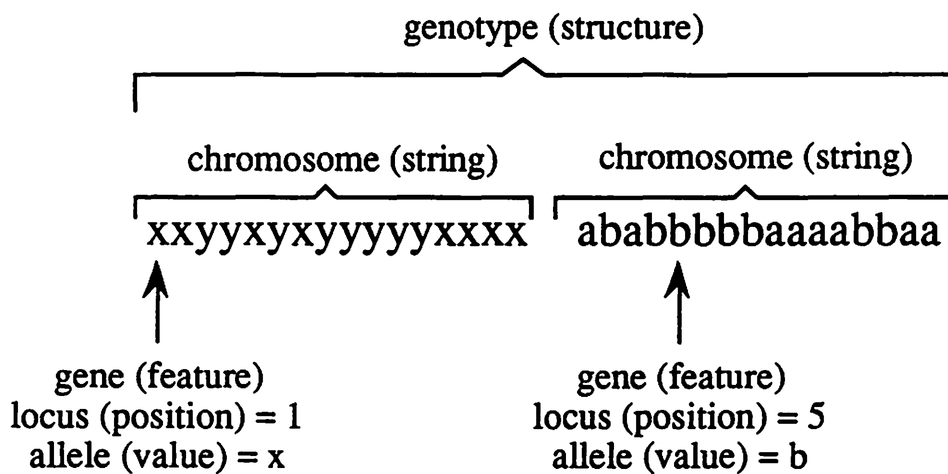


Figure 15. Comparison of natural genetics and genetic algorithm terminology on a representative genotype (structure). Here, a genotype is composed of two chromosomes, each of which has 16 genes.

5.1.3 Genetic algorithm notation

The following notational conventions will be followed throughout the remainder of this thesis when discussing GAs. An upper case letter will be used to denote a chromosome, and a lower case letter will be used to denote a single gene within a chromosome. Thus, a chromosome, A, composed of three genes is represented as $A=a_1\ a_2\ a_3$. An upper case letter with an underscore is used to denote a population. Therefore, $\underline{A}=\{A_1, A_2, A_3,...,A_N\}$ where N is the population size.

5.1.4 Schemata

Before discussing the genetic algorithm operators in detail, it is necessary to describe an important concept in genetic algorithms called schemata [36]. Schemata (plural for schema) are similarity templates which describe a subset of strings with similarities at certain string positions. For example, consider a binary alphabet with an additional symbol, the asterisk, *. The alphabet, denoted as Σ, is now a ternary alphabet and defined as

$$\Sigma = \{0, 1, *\}. \tag{17}$$

The asterisk in this alphabet is treated as a "don't care" symbol, meaning that it will match any of the other symbols in Σ. (The don't care symbol is never actually processed by GAs, but is used merely to illustrate the concept of schemata.) Therefore, the string *00 matches two strings, {100, 000}, while the string *0* can match four strings, namely {000, 100, 001,

101}. H is used to denote a schema. Thus, if H=*0*, then A=101 is one example of the schema H.

For an alphabet with a cardinality (number of elements in the alphabet, not including the don't care symbol) of k, there are $(k + 1)^{\ell}$ schemata, where ℓ is the length of the string. In the previous example, the cardinality of Σ, denoted as $|\Sigma|$, is 2 and the string length is 3, giving the number of schemata as $(2 + 1)^3=3^3=27$. For any particular string, say 101, there are 2^3 schemata represented, {101, 10*, 1*1, 1**, *01, *0*, **1, ***}, because each bit position may take on its own value or a don't care symbol. Thus, when we consider a population of these strings, we can compute lower and upper bounds on the number of schemata contained in any population, for a ternary alphabet as in Eq. 17, as

$$2^{\ell} \leq \text{number of schemata in a population} \leq N \cdot 2^{\ell} \tag{18}$$

where N is the population size. Of course, the number of schemata in any population could be computed exactly by knowing the actual value of each of the strings in the population.

5.1.4.1 Importance of schemata The importance of schemata can best be recognized through an example. Suppose the function to be optimized (the objective function), f, is given as

$$f(x) = x^2. \tag{19}$$

(Although this is a rather trivial optimization problem, it will serve to illustrate several useful points concerning schemata). Suppose further that a population of four genotypes $\underline{A}=\{A_1, A_2, A_3, A_4\}$ are created randomly. Each genotype has a single chromosome consisting of three

genes, a_1, a_2, and a_3. Each of the genes is a binary digit, and the chromosome (also the genotype in this case because each genotype consists of only one chromosome) is decoded as the binary representation of an integer. Thus, the range of values for a genotype is from zero (string 000) to seven (string 111). Table 4 contains the four randomly selected genotypes along with their decoded values, x, and objective function (fitness) values, f(x).

Table 4. Example population of genotypes and objective function (fitness) values $f(x)=x^2$

Genotype Name	Genotype	Decoded Value, x	Objective Function (Fitness) Value, $f(x)=x^2$
A_1	101	5	25
A_2	010	2	4
A_3	110	6	36
A_4	001	1	1

Given only the fitness values for each genotype in a population, how is the next population's overall performance improved? In other words, what information is contained in the genotypes which will help guide the search for more highly-fit genotypes? The answer is schemata.

The important role which schemata plays in guiding a search using GAs is demonstrated in the following way. Consider each of the genotypes in the example as representing the corner of a cube. This cube is shown in Fig. 16 with the genotypes in the population appropriately labeled. Recall that each genotype contains 2^{ℓ} schemata. Looking at genotype A_1, it is seen that A_1 samples not only the single point 101, but also the lines 10*, 1*1, and *01, the planes 1**, *0*, and **1, and the entire search space ***. This last schema provides

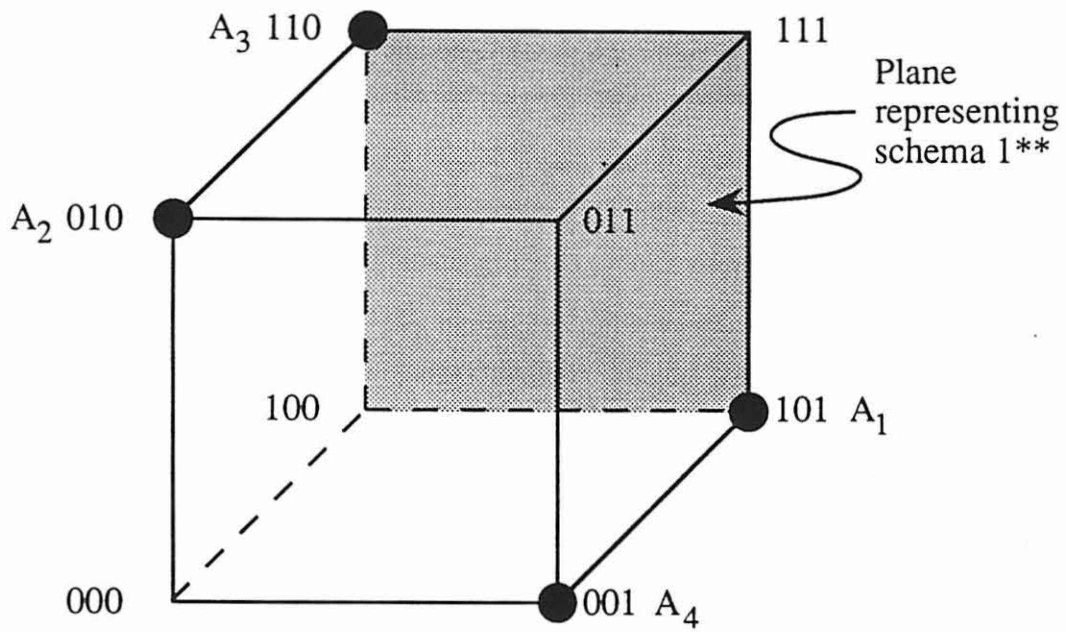


Figure 16. Cube illustrating the role of schemata in genetic algorithms. The large dots at the corners of the cube indicate the genotypes selected in the example (Section 5.1.4.1).

information about the mean of the current population [37]. Because each genotype is able to sample 2^{ℓ} schemata simultaneously, the name intrinsic parallelism is used to describe this behavior [36]. In fact, Holland [36,38] has shown that for a population of N genotypes, on the order of N^3 schemata are usefully processed during each generation!

By simply scanning the genotypes in Table 4, certain similarities can be seen. The most highly-fit strings (those with the highest objective function values) are A_1 and A_3. Both of these genotypes have a value of 1 for gene a_1. Thus, it might be plausible to conclude that the plane containing genotypes with the schema 1** (the back face of the cube, shaded in Fig. 16) is a more highly-fit plane than any of the other planes in the cube. In fact, because of the binary encoding and the objective function used in the example, this is certainly correct. This can be extended further by saying that the line containing genotypes with schema 11* is more highly-fit than any other line in the cube and that the single genotype 111 is more highly-fit than any other point in the cube. Obviously, there must be some mechanisms by which the initial population can be used to create other populations containing genotypes with more of these highly-fit planes, lines, and points. Eventually, of course, the GA should discover the optimal point in the search space, in this case, the string 111. The mechanisms by which this goal can be realized are discussed in Section 5.1.5 and Section 5.1.6 discusses the relevance of schemata in discovering an optimal solution.

This analysis could certainly be extended to hypercubes and hyperplanes for higher-dimensional strings, but graphing such information is difficult.

5.1.4.2 Characterizing schemata There are many different types of schemata. Two devices will be used to characterize a particular type of schema: the schema order and the schema defining length. These will be used in the following sections to evaluate what effect each of the three basic genetic algorithm operators has on schemata in a given population.

Schema order defines the number of fixed positions (positions not occupied by a don't care symbol) present in the template and is denoted as $o(H)$. Therefore, $o(H)$ for H=1*1 is two, or, alternatively, $o(1{*}1)=2$. Similarly, $o(1{*}{*})=1$.

The schema defining length is the distance between the first and last specific (fixed) string positions and is denoted as $\delta(H)$. Thus $\delta(1{*}1)=3-1=2$ and $\delta(1{*}{*})=1-1=0$.

5.1.5 Genetic algorithm operators

Three basic mechanisms, or operators, comprise a genetic algorithm. These three operators are reproduction, crossover, and mutation. Each will be considered separately in the following sections. Reproduction effectively selects the fittest of the genotypes in the current population to be used in generating the next population. In this way, relevant information concerning the fitness of a genotype (and thus information about relevant schemata) is passed along to successive generations. Later (Section 5.1.5.1) it will be shown that GAs actually allocate exponentially increasing trials to the most fit of these genotypes. Crossover serves as a mechanism by which genotypes can exchange information, possibly creating more highly fit genotypes in the process, but most importantly, allowing the exploration of new regions of the search space. Mutation is a secondary operator which ensures that no genetic information is lost forever during reproduction and crossover.

Goldberg [36] gives a detailed mathematical analysis of the effect of each of these three operators on schemata in moving from one population to the next. The next three sections will describe each of the genetic algorithm operators and highlight the analysis presented by Goldberg.

5.1.5.1 Reproduction Reproduction is a mechanism by which the most highly fit genotypes in a population are selected to pass on information to the next population of genotypes. Genotypes are selected in proportion to their fitness with the current population's average fitness. For example, if genotype A_1's fitness value is twice that of an average genotype, say A_2, then genotype A_1 should receive twice as many chances to reproduce as genotype A_2. After a genotype is selected (possibly several times), a copy of it for each instance of being selected is placed into a mating pool along with other highly fit genotypes. Individuals in the mating pool are then paired up and offspring are produced. The offspring of these mates then become the next population. Intuitively, because the fittest of the individuals in the current population are selected in proportion to the average of the population, those individuals possessing good schemata are reproduced in the next generation. This serves to preserve the best schemata found thus far in the search.

What effect does reproduction have on the schemata contained in a given population? Assume at time t that there are m examples of schema H in a population $\underline{A}(t)$, so that $m=m(H,t)$. During reproduction, an individual is selected according to its fitness. Let p_i be the probability of selecting genotype i as given by

$$p_i = \frac{f_i}{\sum_{i=0}^{N} f_i} \tag{20}$$

where f_i is the fitness of genotype i, and $\sum_{i=0}^{N} f_i$ is the sum of the fitnesses for all strings in the population. Then, for the next generation, at time (t+1), the number of examples of schema H, m(H, t+1) is given by

$$m(H, t+1) = m(H, t) \cdot N \cdot \frac{f(H)}{\sum_{i=0}^{N} f_i}. \qquad (21)$$

where f(H) is the average fitness of strings representing schema H at time t, and N is the population size. Now, let $\bar{f}$ denote the average fitness of the population as

$$\bar{f} = \frac{\sum_{i=0}^{N} f_i}{N}. \qquad (22)$$

Substitution of this term into Eq. 21 gives the schema difference equation [36]

$$m(H, t+1) = m(H, t) \cdot \frac{f(H)}{\bar{f}}. \qquad (23)$$

If a schema H remains above average by an amount $c\bar{f}$, where c is a constant, then

$$m(H, t+1) = m(H, t) \cdot \frac{(\bar{f} + c\bar{f})}{\bar{f}} = m(H, t) \cdot (1 + c). \qquad (24)$$

Starting at time t=0, a geometric progression or discrete analog of an exponential form is obtained and given by

$$m(H, t) = m(H, 0) \cdot (1 + c)^t. \qquad (25)$$

This shows that reproduction effectively allocates an exponentially increasing number of trials to the most highly fit individuals (those with above average schemata) in a population.

However, reproduction alone does not serve to investigate new regions of the search space because it only reproduces schemata already present in the current population.

5.1.5.2 Crossover Crossover is the primary genetic operator which promotes the exploration of new regions in the search space. Crossover is a structured, yet randomized mechanism of exchanging information between strings. Crossover begins by selecting, at random, two individuals previously placed in the mating pool during reproduction. A crossover point is then selected at random, and the information from one mate, up to the crossover point, is exchanged with the other mate. For example, consider two genotypes, each consisting of 16 genes. Let A_1=xxyyxyxyyyyyxxxx and A_2=ababbbbbaaaabbaa, and randomly choose the crossover point to be six. Before crossover and showing the crossing point with a vertical bar,

A_1=xxyyxy | xyyyyyxxxx

A_2=ababbb | bbaaaabbaa

and after crossover

A_1=ababbb | xyyyyyxxxx

A_2=xxyyxy | bbaaaabbaa

In this example, there is only one crossover point; more crossover points could have been selected, in which case more information would be exchanged between the two mates. However, as the number of crossover points increases, the probability of breaking good schemata also increases.

Now, the effect of crossover on schemata will be discussed. In Section 5.1.4.2, the definition of a schema defining length was given as the distance between the first and last fixed string positions, denoted as $\delta(H)$. Introducing the crossover operator, the probability of destroying a schema H, $p_d(H)$, is

$$p_d(H) = \frac{\delta(H)}{(\ell - 1)}. \qquad (26)$$

Thus, the probability of survival for schema H, $p_s(H)$, is

$$p_s(H) = 1 - p_d(H). \qquad (27)$$

If the probability of a crossover occurring for any particular mating is denoted as p_c, then

$$p_s(H) \geq 1 - \left(p_c \cdot \frac{\delta(H)}{\ell - 1}\right). \qquad (28)$$

Combining the effects of reproduction and crossover gives

$$m(H, t{+}1) \geq m(H, t) \cdot \frac{f(H)}{\overline{f}} \cdot \left[1 - \left(p_c \cdot \frac{\delta(H)}{(\ell - 1)}\right)\right]. \qquad (29)$$

It is apparent that schemata with short defining lengths are more likely to survive crossover than those with long defining lengths.

5.1.5.3 Mutation The third of the genetic algorithm operators is mutation, and is generally thought of as a secondary operator. Mutation ensures that no string position will ever be fixed at a certain value for all time. Mutation operates by toggling (in a binary alphabet) any given string position with a probability of mutation, p_m. This operator is typically applied to the offspring, after crossover, before completing the generation of the new population. Thus, the probability of survival for a given allele (value) is $(1 - p_m)$.

A given schema H survives only when each of the $o(H)$ fixed string positions within the schema survive; therefore, the probability of a schema surviving mutation is $(1 - p_m)^{o(H)}$. For very small values of p_m, $p_m << 1$, the probability of survival for schema H is approximately $(1 - (o(H) \cdot p_m))$.

Now, combining the effects reproduction, crossover, and mutation, the number of examples of schema H in the next generation is given by

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \cdot \left[1 - \left(p_c \cdot \frac{\delta(H)}{(\ell - 1)}\right) - (o(H) \cdot p_m)\right]. \quad (30)$$

This indicates that low-order schemata are more likely to survive mutation than high-order schemata. The addition of the mutation operator does not alter the effects of reproduction and crossover much. It merely exists to be certain that genetic information is never permanently lost as the search proceeds.

5.1.6 Building blocks

In the preceding sections, it was shown that schemata with above-average fitness values, short defining lengths, and low-order are given an exponentially increasing number of trials as the search progresses. This is a fundamental conclusion in genetic algorithms and is called the Schema Theorem or Fundamental Theorem of Genetic Algorithms [36]. These highly-fit, short, low-order schemata are termed building blocks. Just as a building is constructed from the ground up by using many small, strong bricks, strings in a genetic algorithm are constructed by reproducing short, low-order, highly-fit schemata and exchanging this information between strings. Thus, the best strings in a given population are reproduced and allowed to share, with other highly-fit strings, the information that has allowed these to strings survive.

A brief discussion of these building blocks should prove enlightening. What do these building blocks look like and how do they help to create more highly-fit strings? A graphical technique borrowed from Goldberg [36] will be used to answer these questions.

Recall from the example in Section 5.1.4 that the plane containing genotypes with the schema 1** was the most highly-fit of the planes in the cube. To see how this plane relates to the objective function $f(x)=x^2$, Fig. 17a shows a plot of the objective function overlaid with the schema represented by 1** (shown by the shaded region). It can be seen that the information in this schema captures values of x^2 which are larger than those that fall outside of the shaded region (those represented by the schema 0**). Note that the graphs are plotted from zero to eight, not zero to seven. This is done to aid the visualization of the schema. Therefore, the shaded regions are inclusive on the left side and exclusive on the right side. For example, in Fig. 17a, the shaded region includes the values ranging from four (inclusive), to eight (exclusive). Likewise, plots of schema 1*1 in Fig. 17b, and schema **1 in Fig. 17c show

what information about the objective function is captured by each of these schemata. It should be noted from these figures that each of the schemata captures some information about the fittest string, 111 (the string sought during optimization), and therefore it can be concluded that all three of these schemata must contain important information regarding this string. As counter examples, consider Figs. 17d-f. These figures contain overlays of the schemata, 0**, *0*, and **0, respectively. Note that none of these schemata contain information about the optimal string. This means that the GA is better off searching further with each of the schemata in Figs. 17a-c, than the schemata in Figs. 17d-f. From the earlier analysis, this is exactly what the genetic algorithm is designed to do. Each of the schemata in Figs. 17a-c are highly-fit, possess a short defining length, and are of low-order. Again, this analysis holds for higher-dimensional strings.

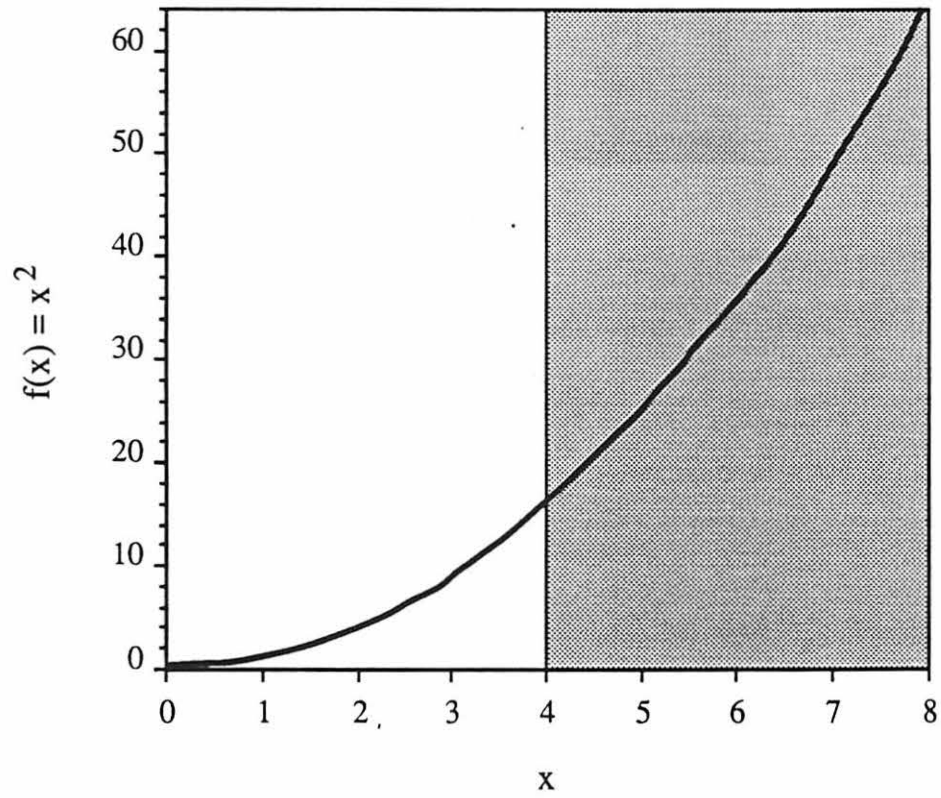


Figure 17a. The objective function $f(x)=x^2$ overlaid with the schema 1**.

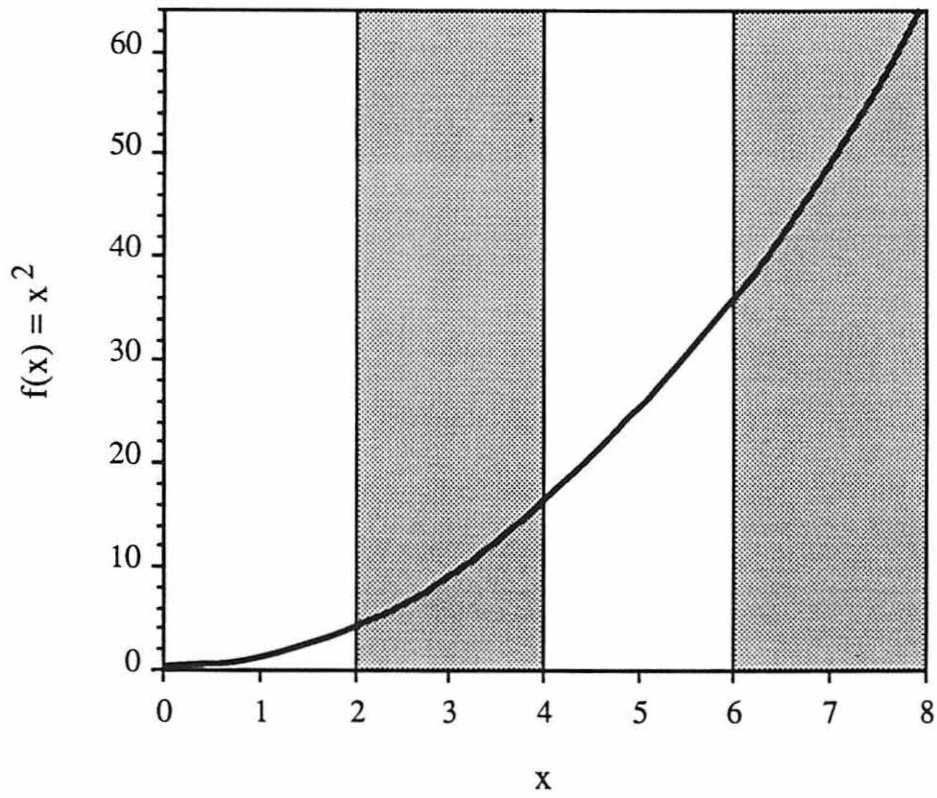


Figure 17b. The objective function $f(x)=x^2$ overlaid with the schema *1*.

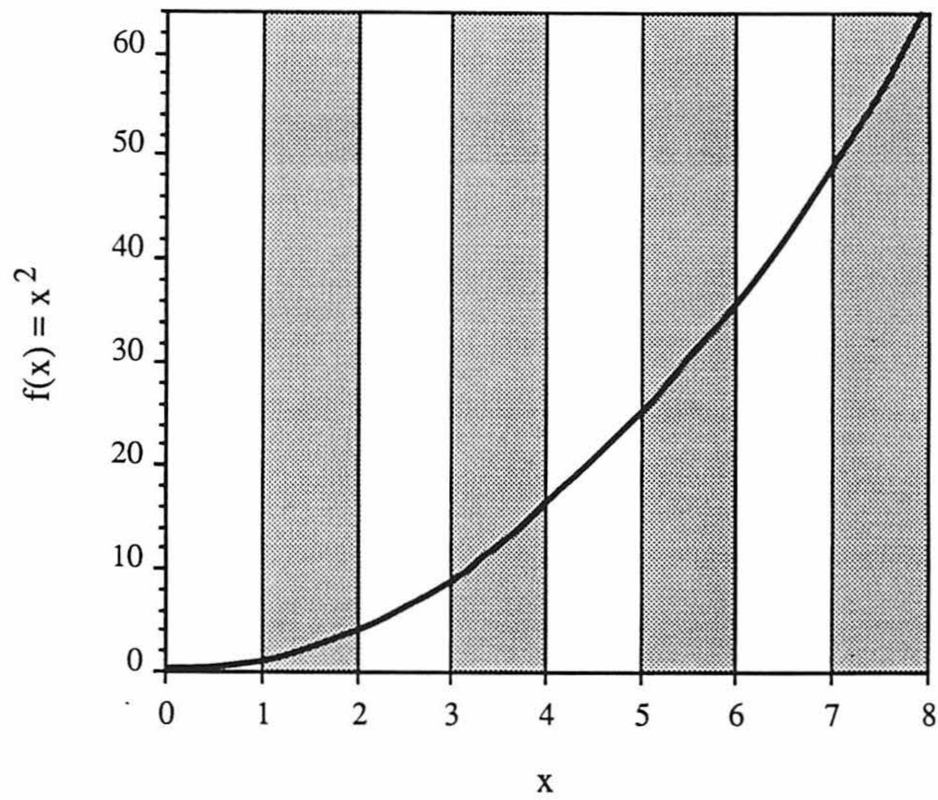


Figure 17c. The objective function $f(x)=x^2$ overlaid with the schema **1.

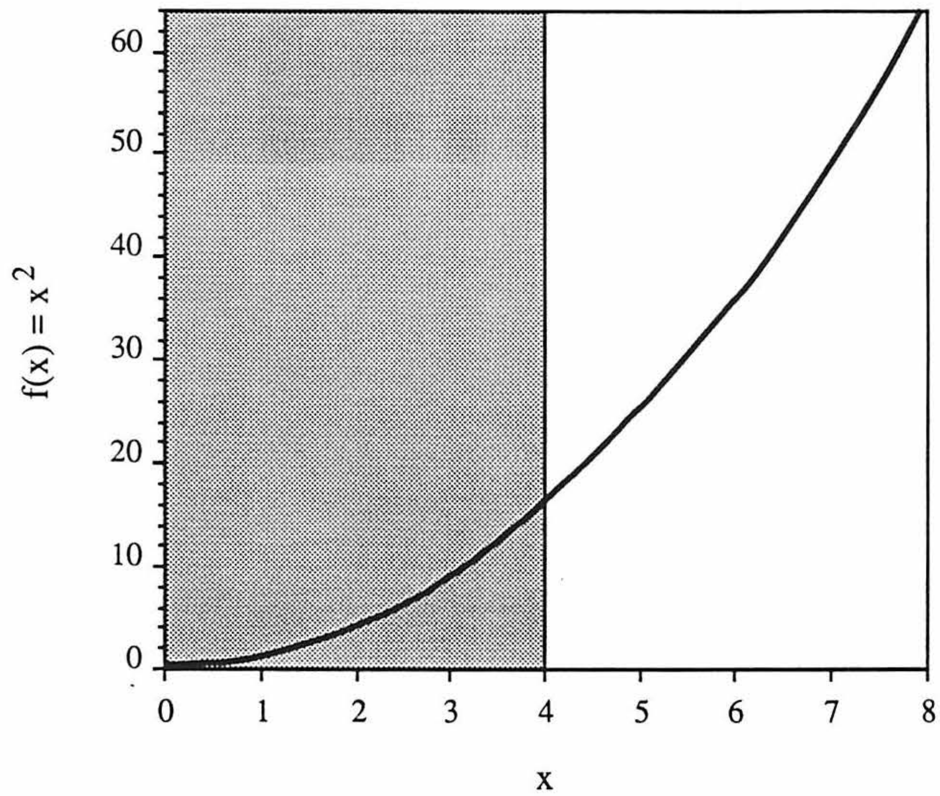


Figure 17d. The objective function $f(x)=x^2$ overlaid with the schema 0**.

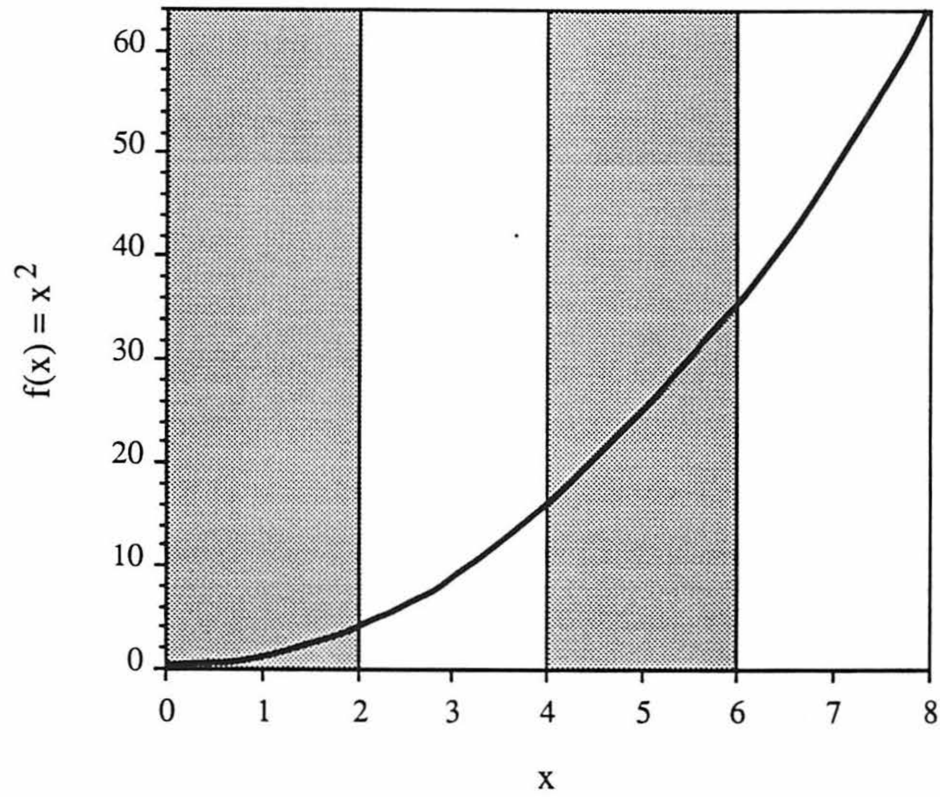


Figure 17e. The objective function $f(x)=x^2$ overlaid with the schema *0*.

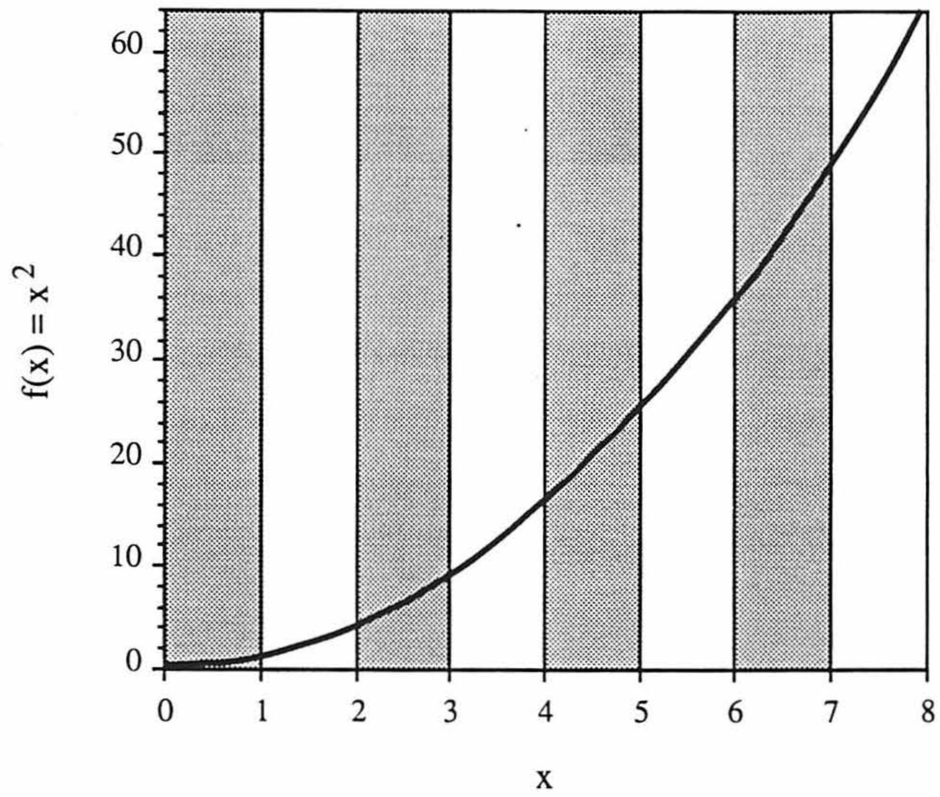


Figure 17f. The objective function $f(x)=x^2$ overlaid with the schema **0.

5.2 Application of Genetic Algorithms to the Optimization of Weights in Neural Networks

As discussed in Chapter 2 and Chapter 3, the backpropagation learning algorithm has several disadvantages associated with it. These include local minima, activation function restrictions (the activation function must be continuous and differentiable), lack of generality for arbitrary network architectures, and long training times for very complex problems, due to the iterative, single-threaded nature of the algorithm. This section introduces the application of genetic algorithms to the optimization of weights in neural networks. This genetic-based learning (GBL) algorithm has the capability of solving all of these problems.

5.2.1 Chromosomes

First, the information to be optimized must be properly encoded so that an appropriate representation may be developed. Because the goal of genetic-based learning, as presented in this chapter, is to optimize the weights in a neural network, each weight in a network must be encoded. Therefore, I will let each weight be binary encoded with a number of bits, b. Each of these weights represents a chromosome. Decoding the chromosome is done by first computing the integer representation of the binary encoding and then mapping this decoded value into a predefined weight range. For example, if eight bits are used to encode a weight, then there are $2^b=2^8=256$ possible decoded values which the weight may have. Suppose a range of ± 8.0 is chosen for the weights. A decoded weight parameter would then be mapped into this range by

$$w = \frac{w_H - w_L}{2^b} \cdot p + w_L \tag{31}$$

where w is the final analog weight value, w_H is the highest possible weight value (+8.0), w_L is the lowest possible weight value (-8.0), b is the string length (8 bits), and p is the decoded parameter. w would then be the value used for the weight represented by its corresponding chromosome in the network. For example, suppose the string representing weight w_1 is 10110100. The decoded value of this string, p, is 180, and the actual weight value would be

$$w_1 = \frac{8.0 - (-8.0)}{2^8} \cdot (180) + (-8.0) = 3.25. \tag{32}$$

Note that there is no longer a smoothly varying function of the weights (as in backpropagation) because the weights can take on only discrete values, dictated by b and the range of weights, determined by w_H and w_L. This implies a resolution of the weights, w_r, of

$$w_r = \frac{w_H - w_L}{2^{\ell}}. \tag{33}$$

Therefore, the solution found by the algorithm will be limited by this weight resolution discretization. In this case, with w_H=+8.0, w_L=-8.0, and using eight bits to encode each weight, there is a resolution of w_r=0.0625. However, this resolution can be made as fine as desired by simply increasing the number of bits used for the encoding.

5.2.2 Genotypes

Next, the genotypes must be defined using the chromosomes. A genotype for genetic-based learning will be the concatenation of all weights in a given network. For example, using the network created to solve the XOR problem in Fig. 4 (Section 2.2), there are a total of nine weights in the network--three each on the two hidden neurodes and three on the output neurode. In this case, a genotype would consist of nine chromosomes as defined in Section 5.2.1. Using eight bits to encode each weight, there would be (8)(9)=72 bits in a genotype.

5.2.3 Population

A population is a collection of genotypes which are evaluated simultaneously during the search. For example, there may be 100 genotypes in a population at time t, all of which have their objective functions evaluated on the basis of their decoded parameters (weights) and the training set at time t. Reproduction, crossover, and mutation then take place on the current population. These three operators (Section 5.1.5.1 - Section 5.1.5.3) produce 100 new genotypes, which are considered to be the next population, at time (t+1). The process of creating a new population of genotypes from the current population continues until the search is completed.

5.2.4 Algorithm

Having discussed the genetic algorithm operators in Section 5.1.5 and having established the coding of chromosomes and genotypes, the algorithm for optimizing weights in neural networks can now be developed. First, for a given problem, the size of the network is specified. For purposes of this thesis, all of the networks will have an input layer, one hidden layer, and an output layer, fully connected in a feed-forward manner. Next, a population of genotypes is created. A single genotype is created by randomly creating a chromosome for each weight in the network. Each member of the population is then evaluated by running the specified network architecture in a forward manner for all pairs in the training set and computing the mean squared error (MSE) for this genotype. The definition of the MSE, as used in my simulations, is presented in Section 5.2.7 when discussing the fitness value of a genotype for genetic-based learning. For a given generation (a generation being a time step), this evaluation is done for all of the genotypes in the population. Thus, there is now a fitness value for each of the genotypes in the population. These fitness values are used for reproduction, crossover, and mutation, as discussed in Section 5.1.5. A new population of genotypes has now been created, and the process starts over by evaluating each of the genotypes using the training set. For clarity and consistency, a flow graph of this process is presented in Fig. 18.

5.2.5 Genetic diversity

An important detail which has been neglected thus far is that of genetic diversity. The term genetic diversity refers to the similarity (or dissimilarity) between genotypes in a given

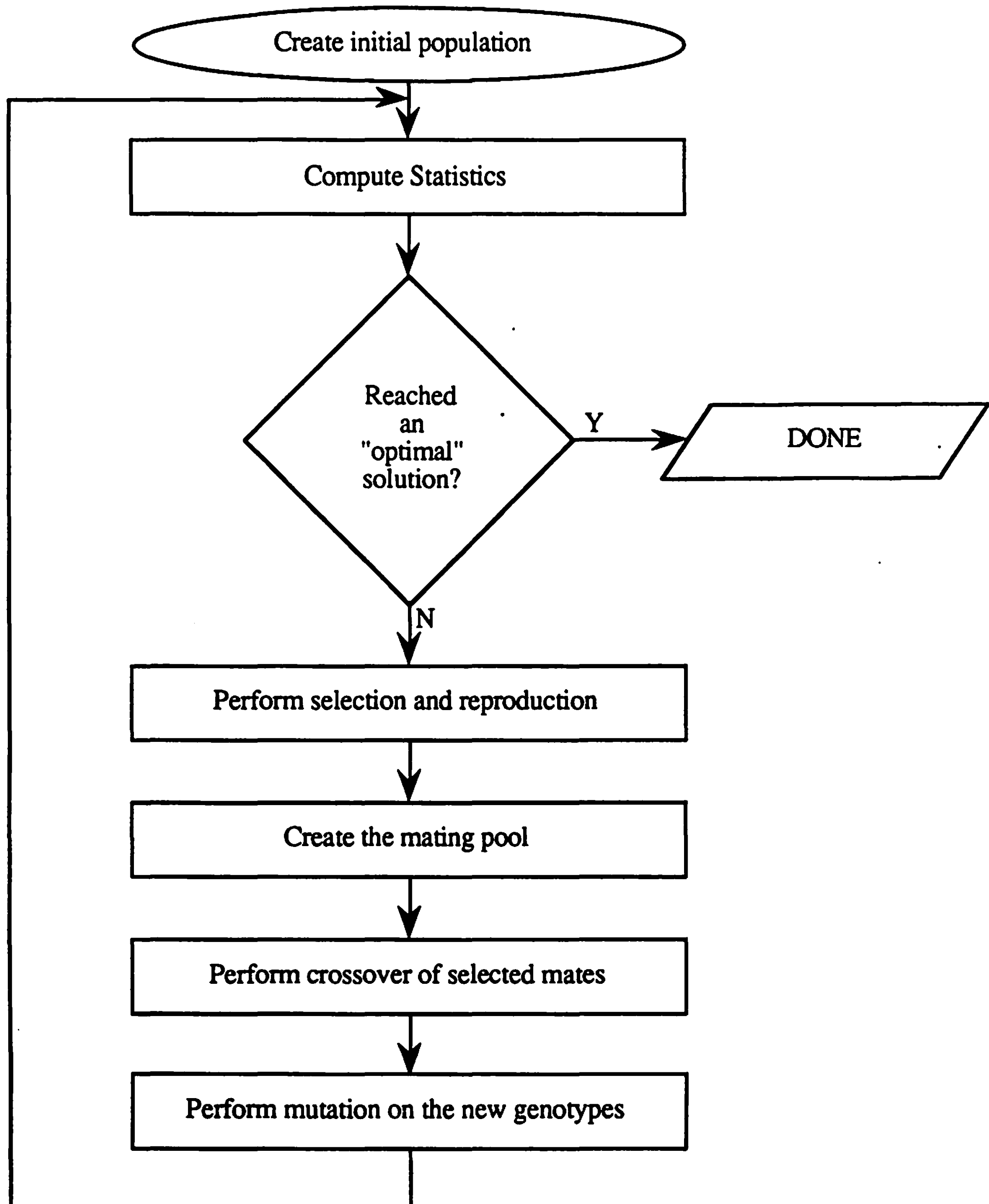


Figure 18. Genetic-based learning flow chart.

population. For example, if all of the genotypes in a population have the same chromosome value for each chromosome, then there is no genetic diversity--all genotypes have the same fitness value because they are all exactly the same. When this happens, all genotypes have an equal chance of reproducing, the mating pool is filled with genotypes which all represent the same network, and crossover alone will no longer be able to sample new hyperplanes. The only method left for exploring new regions of the search space is that of mutation, and if the rate of mutation is very small (much less than unity), then the search will have effectively stagnated due to lack of genetic diversity. Ideally, genetic diversity should remain high, allowing for the investigation of new solutions through the primary operation of crossover. Mutation should only serve to be certain that no gene allele is ever fixed for all time. Several parameters (Section 5.2.6), as well as several improvements to the basic genetic-based learning algorithm (Section 5.3) can help to prevent a lack of genetic diversity.

5.2.6 Parameters

Just as learning constants α and η must be appropriately chosen in the backpropagation learning algorithm, there are several parameters which must be selected in genetic-based learning. The following will discuss each of these parameters separately and suggest typical values selected in my simulations.

5.2.6.1 Population size, N   The size of a population for a given problem can be a crucial factor maintaining genetic diversity and, thus, in finding an optimal solution. A larger population implies increased genetic diversity by sampling more points in the search space simultaneously. However, a larger population also requires more function evaluations during

each generation. For example, a population of 100 genotypes requires 100 forward passes through the network for each input-output pair in the training set. Therefore, training a network to perform the XOR problem, would require 400 forward passes per generation. If the population size is increased to 1000 genotypes, then 4000 forward passes per generation would be required. Clearly a compromise between genetic diversity and search time must be made. As a rule of thumb I generally choose to use between 4 and 10 genotypes for every input-output pair in the training set and for every output neurode. Thus, for the XOR problem with four input-output pairs and a single output neurode, I would use between 16 and 40 genotypes. For a problem such as a two-bit adder (Section 6.4.4) with 16 input-output pairs and three output neurodes, between 192 and 480 genotypes would typically be used.

5.2.6.2 Crossover rate, p_c From experience in running my simulations, the crossover rate, p_c, plays a less important role in finding an optimal solution than does the population size. This parameter controls how often a pair of mates selected during reproduction actually exchange information during crossover. For instance, if p_c=1.0, then every mating exchanges information between the mates during crossover over. However, if p_c=0.5, then on average, only one half of the matings will exchange information.

The crossover rate has several effects. First, a higher crossover rate means that more information is exchanged between genotypes during each generation, implying that more of the search space may be investigated during each generation. Second, as the search progresses and the average fitness of the population increases, a higher crossover rate will tend to destroy highly fit genotypes by splitting good schemata. I have found that a crossover rate of 0.5 works well and use this as a default in my simulations. Other rates have been explored, but using 0.5 seems to work well over a variety of problems.

5.2.6.3 Number of crossover points When performing crossover, more than one crossover point may be selected. Doing so increases the amount of information that is exchanged between genotypes each generation. However, selecting too many crossover points increases the probability of breaking good schemata (see Section 5.1.5.2). All of my simulations use a single crossover point.

5.2.6.4 Mutation rate, p_m As discussed in Section 5.1.5.3, mutation is a secondary genetic operator serving to ensure that no bit position is ever fixed for all time. It should have only a minor impact compared to the primary operators of reproduction and crossover. Typical values of the mutation rate, p_m, used in my simulations range from 0.2 to 0.0001. Later, an adaptive mutation operator will be presented which can help to maintain genetic diversity, especially when using smaller populations. This adaptive mutation operator can actually play a much larger role in exploring the search space than using a non-adaptive mutation operator and, therefore, behaves with properties similar to a primary operator.

5.2.6.5 Chromosome string length, b Chromosome string length refers to the number of bits, b, used to encode each weight in the network. b determines the number of discrete values which any weight can take on during the search. For the binary encoding used here, this number of discrete values is 2^b. Again, there is a tradeoff when choosing b. Larger values of b give a more smoothly varying function of the weights due to the finer granularity of weight values. However, larger values of b also require more steps when processing the genotypes during reproduction, crossover, and mutation. Typical values of b in my simulations are 8, 16, and 32.

5.2.6.6 Weight range, w_H and w_L The weight range specifies the maximum (w_H) and minimum (w_L) values which each weight in the network may have. Therefore, a range which is too narrow can limit the number of possible solutions or even eliminate the optimal solution and thus prevent the network from converging to an "optimal" or acceptable solution. The weight range also determines the resolution of the analog weight values, as discussed in Section 5.2.1. Typical weight ranges in my simulations include ±16.0, ±32.0, and ±64.0.

5.2.7 Fitness value

In order to select individuals to be placed in the mating pool during reproduction, it is necessary to evaluate each of the individuals based on some criterion. This criterion is known as the objective function, which produces a fitness value. Because the goal of genetic-based learning in neural networks is to optimize the weights of the network, each individual must be evaluated on the training set. The typical measure of a network's performance on the training set is the Mean Squared Error (MSE) given by

$$MSE = \frac{1}{T} \sum_{t=1}^{T} \sum_{k=1}^{K} (y_{tk} - \hat{y}_{tk})^2 \tag{34}$$

where T is the number of input-output pairs in the training set, K is the number of output neurodes, y_{tk} is the target value for the kth output neurode, and $\hat{y}_{tk}$ is the network estimate of y_{tk}. In order to increase selective pressure for networks with multiple output neurodes, I have modified this MSE slightly by dividing not by T, but by TK. Therefore, as implemented in my GBL algorithm, the MSE_{GBL} is

$$MSE_{GBL} = \frac{1}{TK} \sum_{t=1}^{T} \sum_{k=1}^{K} (y_{tk} - \hat{y}_{tk})^2. \qquad (35)$$

The MSE_{GBL} will serve as the objective function for the genetic-based learning algorithm. However, because genetic algorithms generally work as function maximizers, the fitness value for a genotype is the inverse of MSE_{GBL}.

5.3 Genetic-Based Learning Enhancements

A comparison of backpropagation learning and genetic-based learning is given in Chapter 6. That chapter also includes a discussion of the computational requirements of each of the algorithms. This section will present several additions and variations made to the standard genetic-based learning algorithm presented earlier. These variations are designed to foster a better search, while at the same time reduce the number of generations needed to reach an acceptable solution. Not only is backpropagation compared with GBL, but the standard GBL algorithm is compared with various algorithms employing the following enhancements. The results in Chapter 6 will clearly show the necessity of these learning enhancements.

5.3.1 Elitist model

One of the problems encountered early in my simulations using the standard genetic algorithm operators was that the fitness value of the best individual in the population often

fluctuated. By this I mean that in moving from one generation to the next, the highest fitness value in the population might actually decrease, rather than increase. A typical cause of this was that the best solution found up to the current population would be lost due to sampling error, crossover, mutation, or some combination of all three. Further, these operators did not produce an individual of equal or greater fitness during this generation. The elitist model [39,40] is meant to remedy this problem.

In essence, the elitist model ensures that the best individual is never lost in moving from one generation to the next, unless a more superior individual is created through the GA operators. In practice, my implementation first performs all of the reproduction, crossover, and mutation operators as specified above. Then, if the most fit individual created by these operations is weaker than the previous most highly fit individual, the weakest individual in the newly created population is replaced by a duplicate of the previously most fit individual. It may happen (and hopefully so) that an individual is created by other matings which has a better fitness than that which was the previous best, in which case, this new individual becomes the most fit in the next population and is consequently preserved during the next reproduction, crossover, and mutation period, if necessary.

In Chapter 6 it will be seen that using an elitist model can have a very positive effect on the number of generations required to find an acceptable solution.

5.3.2 Sliding window to induce selective pressure

Another problem that can occur in using genetic algorithms is associated with selective pressure. Selective pressure relates to the manner in which individuals are selected for reproduction. Although genotypes are selected in accordance with their fitness value relative to

the rest of the population, it can often happen, especially during the initial stages of the search, that all of the individuals have fitness values which are nearly the same. I found this to be especially true when training larger networks and networks with multiple output neurodes. The latter case is the reason for using a modified MSE measurement as given in Section 5.2.7. When all (or nearly all) of the individuals have similar fitness values, the search can stagnate because those individuals which possess even slightly higher fitness values may not, due to the probabilistic nature of the selection mechanism, receive more chances during reproduction than those with slightly lower fitness values. For example, suppose that at a certain generation, all of the genotypes have fitness values, f(x), in the range $45 < f(x) < 55$. It is clear there is not much deviation among the genotypes. However, if a new parameter, f_{min}, is defined to be 40, and this value is subtracted from all of the genotypes' fitness values before selection, then a genotype with a value of 55 originally would now appear to have fitness which is three times as great as a genotype with a value of 45 originally. If this scaling is not performed, then the relative difference between these two genotypes would be about 1.2 times; a small relative difference which can easily become lost due to sampling error.

In my simulations, I chose to use a scaling window of one, meaning that the minimum fitness value from the population which immediately preceded the current population is used as the value of f_{min}. Another nice feature of using a sliding window to help induce selective pressure is that it will typically have more effect during the initial stages of the search when all of the networks have random weights, and, therefore, nearly the same fitness values. As the search progresses, the minimum fitness value will commonly remain quite low with respect to the average and best individuals. If this is the case, then the sliding window will have little effect on the rest of the search because subtracting this minimum value will not change the relative fitness values between the individuals much. The sliding window tends to provide more pressure at the beginning of the search where it is needed, yet not interfere with the

search in the later stages where one would like to see only the genetic algorithm operators play the major roles.

This enhancement mechanism is used where noted in the following chapters.

5.3.3 Adaptive mutation operators

Another problem often encountered, and discussed briefly in Section 5.2.5, is that of a lack of genetic diversity. With the selection mechanism used in my simulations, if an individual which is much superior to the rest of the population is created, then that individual will often receive an abundance of reproductive trials when generating the next population. In fact, if the individual is very much superior, it could happen that the entire mating pool consists only of copies of this individual. When this happens, that individual dominates the population and the search will stagnate because new regions of the search space can not be explored by the primary operator, crossover. This phenomenon is termed a lack of genetic diversity.

Several methods for maintaining genetic diversity have been explored [37,41,42]. The one I chose to implement is a variation of one proposed by Whitley and Hanson [37] which I call a weighted Hamming distance (WHD) adaptive mutation operator. I will first discuss the operator employed by Whitley and Hanson and then follow with a discussion of several variations as devised by me.

The adaptive mutation operator used by Whitley and Hanson adjust the probability of mutation dependent upon the genetic diversity of the current population. Therefore, there is no longer a constant mutation rate, but rather one that varies within a predetermined range. In order to adjust the mutation rate, a method of measuring genetic diversity must be developed. As a heuristic, the distance between two genotypes during a mating is calculated. This distance

measurement is designed to determine the similarity between two genotypes. Measuring the distance between two genotypes provides a rough measure of the diversity of the population. The method used in calculating the distance can produce significantly different results, and this is where I propose several alternative methods to the one employed by Whitley and Hanson. Whitley and Hanson chose to use a direct Hamming distance (HD) measure, where the Hamming distance is defined to be the number of bits in which to strings differ. The lower this Hamming distance, the higher the mutation rate. A lower Hamming distance implies that the two mates are very close to one another in bit-wise manner, and therefore are relatively closely related in their weight values.

Determining the actual probability of mutation for any mating is a simple exercise. First, an allowable range for the mutation probability is selected, given by the upper bound of mutation, p_{mu}, and lower bound of mutation, p_{ml}. If the Hamming distance between two genotypes during a mating is HD, then the probability of mutation, p_m, is given as

$$p_m = \left[(p_{mu} - p_{ml}) \left(\frac{HD_{max} - HD}{HD_{max}} \right) + p_{ml} \right]^2, \qquad (36)$$

where HD_{max} is the maximum Hamming distance which could occur between any two genotypes. A squared adaptive mutation probability is used because it was found by Whitley and Hanson that this tended to work better than a linear operator. I have also found similar results. Using Whitley and Hanson's method, HD_{max} is equivalent to the total length of a genotype, given by

$$HD_{max} = b\, W_T, \qquad (37)$$

where W_T is the total number of weights in the network.

For example, if eight bits, b, are used to encode any weight in a network, the range of weights (w_H and w_L) is ±8.0, and a genotype is represented by two weights, where genotype one, G_1, is 0010110111001101, and genotype two, G_2, is 0010110011001100, then by Eq. 31, the first weight for G_1, w_{11}, is -5.1875 and the second weight for G_1, w_{12}, is 4.8125. Likewise, the first weight for G_2, w_{21}, is -5.25 and w_{22}, is 4.75. In this case W_T=2, HD_{max}=(8)(2)=16, HD=2, and let p_{mu}=0.3 and p_{ml}=0.001, then p_m=0.0689712. The difference between these two genotypes (noted by the underlined bits), both in actual number of differing bits and the analog weight values represented, is small, meaning that the two networks represented by these genotypes are essentially similar. Because the networks represented by these two genotypes are very similar, increasing the probability of mutation will help to explore new regions of the search space. Crossover alone on these two genotypes will not produce new genotypes which substantially vary from the two mates, thus not sampling different regions. Now, if G_2 is changed to be 1010110101001101, then w_{21}, is 2.8125 and w_{22}, is -3.1875. Even though the two genotypes differ by only two bits, the difference in the analog weight values represented by the genotypes is significant. A Hamming distance measurement would still measure this distance as being two, the same as in the previous, and would maintain a relatively high probability of mutation even though the networks represented are quite different.

When using binary encoded parameters as done in genetic-based learning, a direct Hamming distance measure does not accurately reflect the distance by which two strings differ. A one bit difference in a binary encoded parameter means much more when it occurs in the most significant digit (MSD) than it does when it occurs in the least significant digit (LSD). Therefore, I propose using a weighted Hamming distance measure which accounts for the position in which bits differ. I have experimented with three methods of weighting the position

of bit difference. These methods are linear weighting (LinWHD), $\log_2$ weighting (LogWHD), and power weighting (PowWHD). Each of these modifies only HD_{max} and HD in Eq. 36.

A linear weighting increases the Hamming distance measure by a linear amount proportional to the position in which it occurs within a weight parameter. In my implementation, a difference in the LSD of a weight increases HD by one, a difference in the next LSD increases HD by two, and so, until the MSD, in which HD is increased by the length of the weight parameter string. Therefore, the maximum HD possible between two genotypes is given by

$$LinWHD_{max} = (b+1)\left(\frac{b}{2}\right)W_T. \tag{38}$$

For a linear weighting method, in the first example above, HD=2 and $LinWHD_{max}$=72, giving p_m=0.0850856. In contrast, for the second example, HD=8+8=16 and p_m=0.054548. Because the difference in actual analog weights for the second example is greater than for the first example, there is actually more genetic diversity in the population in the second example, and therefore, the mutation probability should be lower. This is represented in using the linear weighting method.

The $\log_2$ weighting method is similar to the linear weighting method, but on a $\log_2$ scale. If β represents the position in which the bits differ within a weight, then as the position increases in significance (moves from LSD to MSD), HD is increased by $\log_2\beta$. Here,

$$LogWHD_{max} = W_T \sum_{i=1}^{b} \log_2(i). \tag{39}$$

In the first example above, $LogWHD_{max}$=30.598416, HD=$\log_2(1)+\log_2(1)=0$; therefore, p_m=0.090. In the second example, HD=$\log_2(8)+\log_2(8)=6$ and p_m=0.0582592. Notice that a $\log_2$ weighting method maintains a relatively higher mutation probability than the linear weighting method.

The third and final method of weighting the Hamming distance is a power weighting. This method is based on increasing HD by 2 to the power of the position, β, in the which the bits differ. In this method

$$PowWHD_{max} = W_T \sum_{i=0}^{b} 2^i. \tag{40}$$

Again, using the first example above $PowWHD_{max}$=1022, HD=$2^1+2^1=4$, giving p_m=0.089299. In the second example, HD=$2^8+2^8=512$ and p_m=0.0225623. For convenience, Table 5 compares the value of p_m for each of the adaptive methods.

Table 5. Comparison of adaptive mutation operators on the two examples presented in this section

Adaptive operator	HD_{max}	Distance for example #1	p_m for example #1	Distance for example #2	p_m for example #2
HD	16	2	0.0689712	2	0.0689712
LinWHD	72	2	0.0850856	16	0.0545480
LogWHD	30.598416	0	0.0900000	6	0.0582592
PowWHD	1022	4	0.0892990	512	0.0225623

Which, if any, of these adaptive mutation methods is the best? After running many simulations, one method still does not stand out clearly as "the optimal" adaptive mutation method for all problems. However, in Chapter 6, I will compare two of these methods on a suite of binary test mappings. These two methods are the unweighted HD method and the power weighting HD method. It is enlightening to visualize the difference between these methods when selecting a probability of mutation for two genotypes during a mating. Consider two four-bit strings, in which one string is held constant, at a value of 0000 (decoding to a value of zero), while the other string is varied between zero (0000) and 15 (1111). Figure 19 shows the result of plotting the value of p_m for each of the adaptive mutation methods versus the number represented by decoding the second string. Note that the power weighting method provides a smoothly varying function of p_m as the difference in analog weight value increases. Clearly this is closest to what is desired. As the difference in the decoded parameters increases, p_m decreases. In contrast, with the other three methods, the value for p_m can jump up and down, quite dramatically in some cases.

It was noted in Section 5.2.5 that one method of helping to maintain genetic diversity was using a large population. However, this also involves more computation in evaluating each population. With these adaptive mutation operators, the population size can be reduced, yet still maintain good genetic diversity [37]. In Chapter 6, however, I will use the same number of individuals for each of the test cases in order to be able to compare performance more directly.

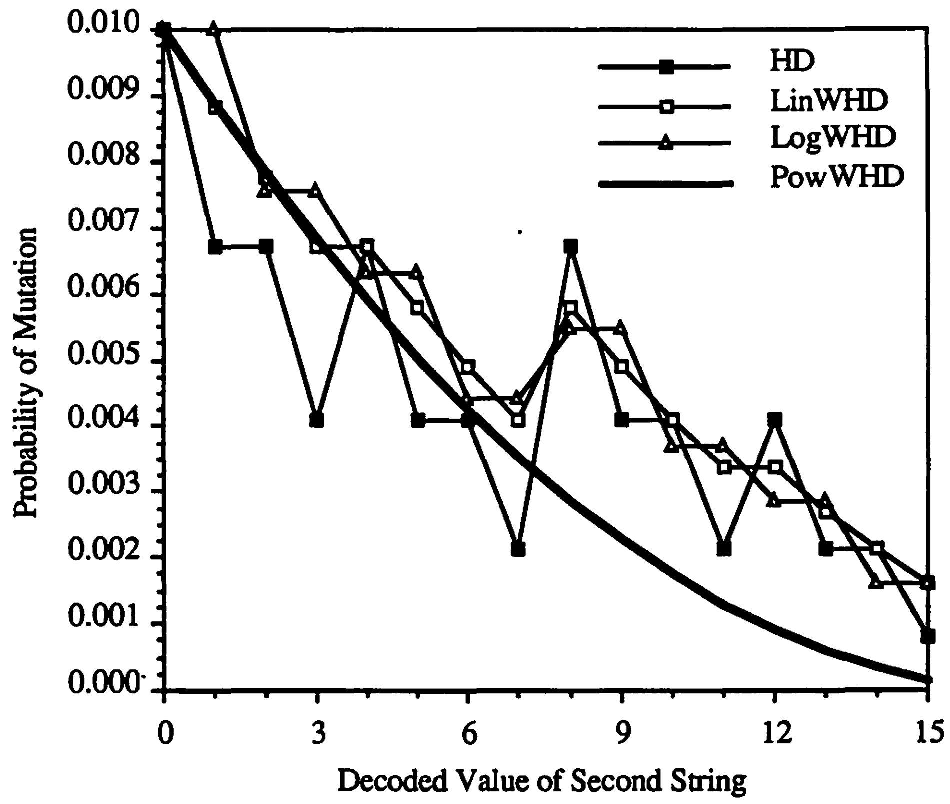


Figure 19. Comparison of adaptive mutation methods on two four-bit strings.

5.3.4 Parallel implementation

Perhaps the greatest improvement that can be made to the genetic-based learning algorithm is to parallelize it. It may be apparent by now that a great deal of the computation involved in performing a genetic search is done by operating on individuals or pairs of individuals. In fact, the only operations which involve considering the entire population simultaneously are that of selection, reproduction, and shuffling to create the mating pool. Selection operates only on the computed fitness values, not on the genotypes directly, reproduction involves simply copying the individuals' indices into a temporary array, and shuffling is a randomization of this array. All three of these operations are quite simple and can be implemented efficiently. The remaining operations, objective function evaluation (operating each of the networks in a feed-forward manner over the entire training set and computing the MSE), crossover, and mutation are all performed on single individuals or pairs of individuals. The objective function evaluation is the most time consuming of these operations, particularly for large mapping problems and/or large networks. Crossover involves bit copies and swaps, and mutation involves bit operations (bit toggles) on individuals when necessary.

Therefore, much of the work done by the genetic-based learning algorithm can be performed in parallel by splitting the population into disjoint subpopulations and operating in parallel on these subpopulations where appropriate. Selection, reproduction, and shuffling must be performed by a single processor. This merely involves synchronizing the parallel processes through a signalling mechanism. Figure 20 is a flow chart illustrating this parallel implementation.

All of the results for the GBL simulations presented in Chapter 6 were accomplished using this parallel algorithm. Chapter 6 will also discuss the implementation of this algorithm.

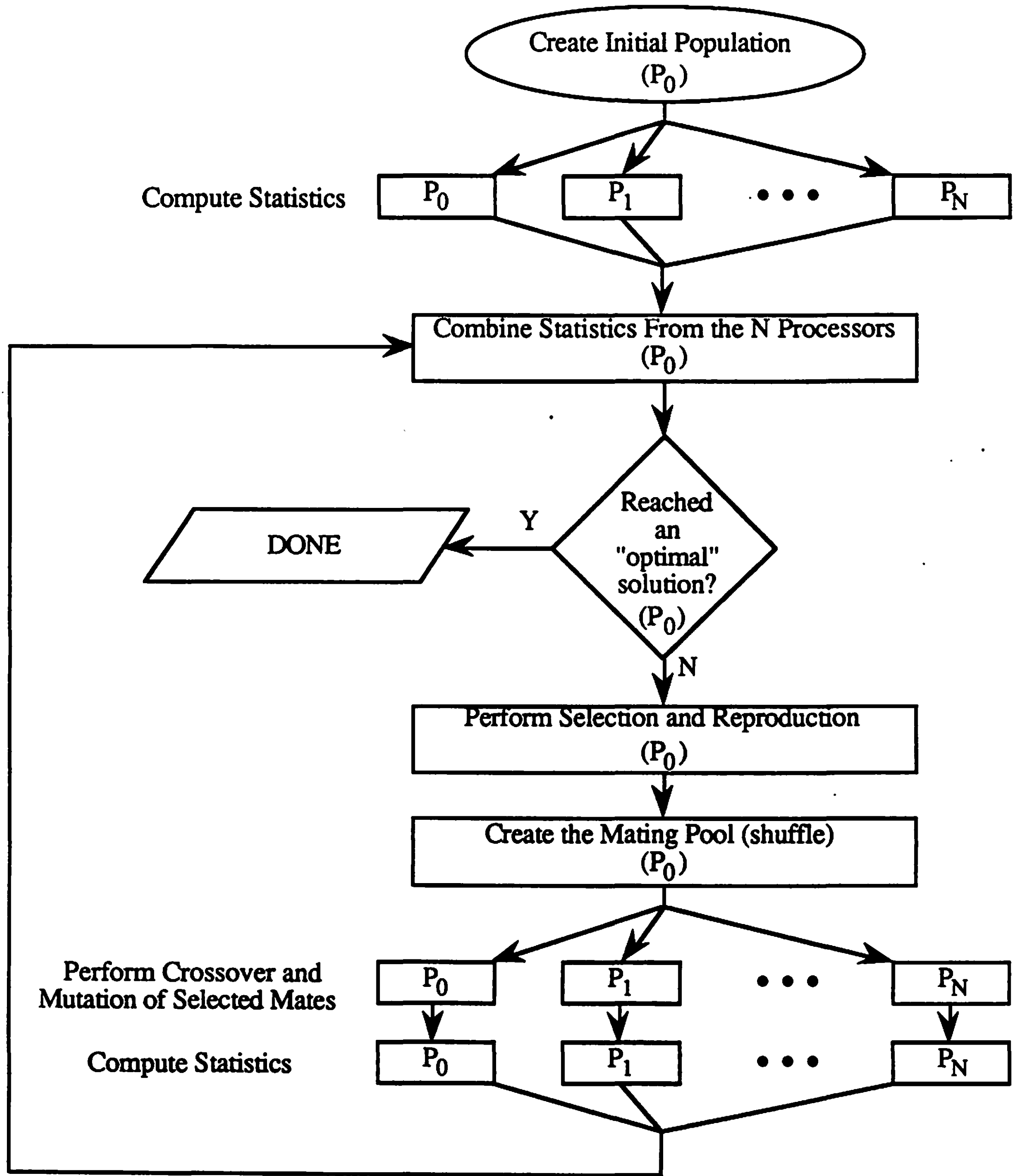


Figure 20. Flow chart of the parallel implementation of genetic-based learning.

It should be noted that I have been able to achieve linear performance improvement over a purely serial algorithm using this parallel algorithm.

5.4 Conclusions

This chapter has introduced a learning algorithm, genetic-based learning, which uses the operations and mechanisms observed in natural selection and genetics. This learning algorithm has the potential for solving several of the disadvantages associated with the backpropagation learning algorithm (Chapter 3), most notably avoiding local minima, being robustly applicable to networks with arbitrary architecture and topology, and to neurodes which have activation functions which may not be differentiable, or for which it may be costly to compute its derivative. Further, and perhaps most importantly from a learning time standpoint, this learning algorithm is quite naturally and easily implemented on massively parallel computers. The parallel algorithm developed in Section 5.3.4 is designed for a shared-memory, multi-processor machine, and I believe this type of machine is best suited for the implementation presented in that section. A message-passing machine would involve far too much overhead in order to be able to achieve the linear performance improvement I have seen.

Genetic-based learning, though, does have some drawbacks. First, even when implemented in parallel, the time required for evaluating a population on very large problems and networks may become unacceptable, although to date this has not been the case. However, if, as will be shown in Chapter 6, genetic-based learning requires fewer iterations (generations) than backpropagation learning, then this may not pose a problem. Second, on even modestly sized networks, this algorithm uses a large amount of memory. Scaling the size of the networks up increases the memory usage by an amount proportional to the population

size. Unfortunately, as the size of the network and the mapping being solved grows, so too must the size of the population.

In all, I have seen very good performance from this learning algorithm and believe it possesses several desirable characteristics which backpropagation lacks. As interest increases and research progresses into using genetic algorithms for optimizing neural networks, some of its limitations and problems will certainly be overcome.

As a concluding remark, I believe that GBL can, and should, be used for purposes other than just optimizing the weights in a neural network. I envision using this learning paradigm to configure the network architecture, the activation functions employed by the neurodes in the network, and the parameters used by the activation functions in computing a neurode's output values, just to name a few.

6 COMPARISON OF BACKPROPAGATION AND GENETIC-BASED LEARNING

6.1 Overview

This chapter presents a comparison of the backpropagation learning algorithm (Chapter 3) and the genetic-based learning algorithm (Chapter 5). This comparison is based on several points. The first is an analysis of the computational and memory requirements for each of the algorithms. Much research has been put forth recently [20-25] in ways to reduce the training time of networks. Most of this research has focused on heuristic and theoretical methods of improving backpropagation by reducing the number of iterations required to obtain a good solution. It will be shown in this chapter that genetic-based learning often requires many fewer iterations (generations) than does backpropagation. Further, while each iteration of the genetic-based learning algorithm requires more processing than an equivalent backpropagation iteration, the genetic-based algorithm is quite naturally and easily implemented in parallel with linear (or near linear) speed up.

6.2 Computational Requirements of the Backpropagation Learning Algorithm

Part of the process of evaluating different learning mechanisms is to evaluate the computational requirements of the algorithms. This section presents an analysis of the

approximate number of floating point operations and memory requests required during the backward pass (the learning phase) of the backpropagation learning algorithm. In Section 6.3, an analysis of similar requirements for genetic-based learning will be given. Because both paradigms utilize the same algorithm during the forward pass, the analysis of this pass will be omitted; only the learning phase of each algorithm will be analyzed.

It will be helpful to refer to Fig. 7. I will assume that a momentum term with a batch size of T is being used for backpropagation and that a network with an input layer, a single hidden layer, and an output layer is being used. Let L_I denote the number of neurodes in the input layer, L_H denote the number of neurodes in the hidden layer, and L_O denote the number of neurodes in the output layer. The backpropagation learning phase can be broken down into five steps. Because of the batching property, the first four steps (BPO_1-BPO_4) occur T times for each single change of the weights.

The first step is to calculate error signals for each of the output neurodes. Referring to Eq. 3, for each output neurode, this requires two memory reads (one to recover the output of the neurode and one to get the target value for this iteration for the current neurode), one memory write (to store the result), two subtractions, and two multiplications (recall that the derivative for the sigmoidal activation function can be computed with a single subtraction and a single multiplication as shown in Eq. 13). The number of operations required for this first step, BPO_1, is

$$BPO_1 = T\, L_O\, [\, 2\ \text{read} + 1\ \text{write} + 2\ \text{sub} + 2\ \text{mult}]. \qquad (41)$$

The second step is to calculate and update the change in weights for each of the output neurodes. The update is necessary for batching purposes, which requires a read to recover the accumulated value for the change in weights and an addition of the newly calculated value. For

each output neurode, there are (L_H+1) weights, including the bias weight. Referring to Eq. 4, calculating the change in weights for the current input-output pair involves three memory reads (one for the learning rate α, one for the error signal, and one for the input value on the corresponding weight), one write (to store the result), and two multiplications, in addition to recovering the accumulated value and updating this value, which gives

$$BPO_2 = T\,(L_H + 1)\,[4\text{ read} + 1\text{ write} + 2\text{ mult} + 1\text{ add}]. \tag{42}$$

The third step occurs after moving down to the hidden layer. The error signals for each of the hidden neurodes is calculated. This calculation differs slightly from the output layer error calculation, as shown in Eq. 6. This involves a read to recover the output value of the neurode plus two additional reads for each output neurode, one each for the interconnection weight and the error signal. In addition, for each output neurode one multiplication is required. Finally, one subtraction and two multiplications are needed to complete the error signal calculation. In total,

$$BPO_3 = T\,L_H\,[\,(1 + 2L_O)\text{ read} + 1\text{ write} + L_O\,(1\text{ mult}) + 1\text{ sub} + 2\text{ mult}]. \tag{43}$$

Step four is similar to step two, in that calculating the change in weights involves four reads, one write, one add, and two multiplications, thus

$$BPO_4 = T\,(L_I + 1)\,[4\text{ read} + 1\text{ write} + 2\text{ mult} + 1\text{ add}]. \tag{44}$$

Finally, step five changes all of the weights in the networks. This occurs only once for every T input-output pairs. For each weight, as shown in Eqs. 8 and 9, this requires four reads, one write, two additions, and one multiplication, giving

$$BPO_5 = [L_O (L_H + 1) + L_H (L_I + 1)] \text{ [4 read + 1 write + 2 add + 1 mult].} \tag{45}$$

Combining all five of these steps, the total number of operations required for a single, backward learning pass, BPO_{total}, is

$$\begin{aligned} BPO_{total} = & [L_H(4L_I+4L_O+5T) + 2L_O(2+T+L_HT) + T(8+4L_I)] \text{ read} + \\ & [L_H(1+L_I+L_O+4T+L_OT) + L_O(1+2T) + T(4+2L_I)] \text{ mult} + \\ & [L_H(1+L_I+L_O+2T) + L_O(1+T) + T(2+L_I)] \text{ write} + \\ & [2L_H(1+L_I+L_O) + 2L_O + T(2+L_H+L_I)] \text{ add} + \\ & [T(L_H + 2L_O)] \text{ sub.} \end{aligned} \tag{46}$$

Using the XOR mapping problem as an example with two input neurodes, two hidden neurodes, a single output neurode, and four input-output pairs, BPO_{total}=156 read + 89 mult + 45 write + 38 add + 16 sub. Doing the same for the most difficult of the test cases, adder 2, BPO_{total}=1576 read + 843 mult + 347 write + 262 add + 176 sub.

6.3 Computational Requirements of the Genetic-Based Learning Algorithm

Because genetic-based learning uses mostly memory operations (copies, swaps, toggles, etc.), not floating point operations, for comparison I will assume that M_{SD} memory operations can be performed in the time it takes to perform a single floating point sum or difference

operation and that M_{MD} memory operations can be performed in the time it takes to perform a single floating point multiplication or division operation. M_{SD} and M_{MD} will vary greatly depending on the machine on which the algorithms are run. On a PC machine without a coprocessor, M_{SD} may be as high as 50 or 100, while M_{MD} may be as high as 100 or 500. On a mini or super-mini computer, a floating point operation may be able to be performed in the same amount of time as a memory operation, or nearly so; therefore M_{SD} and M_{MD} may be as low as one.

This analysis will also be broken into two distinct sections, one which involves the operations required to be performed by a single processor (selection, reproduction, and creation of the mating pool), and one which involves operations that can be performed on multiple processors (crossover and mutation).

Let b denote the number of bits used to encode a weight, W_T denote the total number of weights in the network (given by $[L_O(L_H+1) + L_H(L_I+1)]$,) N denote the population size, p_c denote the probability of a crossover occurring for any mating, and p_m denote the probability of mutation for any bit. Creating the mating pool is performed through three simple steps, selection, reproduction, and shuffling. Selection comprises two steps. First, the average fitness of the population must be computed. Again, I am assuming that the forward pass has already been completed, meaning that the MSE_{GBL} has been computed for each individual. This implies N memory reads (to recover the fitness of each individual), (N-1) summations, one division, and one write. Next, the expected value (number of copies of each individual to be placed in the mating pool) is computed, which entails N reads, N divisions, and N writes. Therefore, the number of operations required during selection, $GBLO_{S1}$, is

$$GBLO_{S1} = 2N \text{ read} + (N+1) \text{ write} + (N-1) \text{ add} + (N+1) \text{ div}. \qquad (47)$$

Next, reproduction takes place, requiring copying of the selected individuals' indices into a temporary array, giving

$$GBLO_{S2} = N \text{ read} + N \text{ write}. \tag{48}$$

The final step of the serial section requires shuffling the temporary array to produce the final mating pool for this generation. In the worst case, this shuffling routine requires a number of memory copies given by

$$\sum_{i=1}^{N} i \text{ (read + write)}. \tag{49}$$

This number is based on an algorithm which first selects an element in the original array at random, places this element in a new array, and finally compacts the original array. This process continues until all elements of the original array have been placed in the shuffled array. In the worst case, the first element of the original array is selected each time, in which case compaction of the original array requires compacting the entire original array. In the best case, the last element is selected each time and no compaction is required. Taking the average of these cases requires dividing Eq. 49 by two. First, Eq. 49 can be cast into the following form, allowing for easier calculation:

$$(N + 1)\left(\frac{N}{2}\right) \text{(read + write)}. \tag{50}$$

Dividing Eq. 50 by two and simplifying gives

$$GBLO_{S3} = \frac{N^2 + N}{4} \text{ (read + write).} \tag{51}$$

Combining the terms for calculating expected values, reproduction, and shuffling, gives the number of floating point and memory operations required for the serial section, $GBLO_S$, as

$$GBLO_S = \left(\frac{N^2+13N}{4}\right)\text{read} + \left(\frac{N^2+9N+4}{4}\right)\text{write} + (N-1)\text{ add} + (N+1)\text{ div.} \tag{52}$$

The second section considers the operations that can be performed by multiple processors in a parallel implementation, namely crossover and mutation. During crossover, assuming that any position along an individual's string has an equal chance of being chosen, on average one-fourth of the bits will be swapped between mates with probability p_c. This is because in the worst case, when the crossover point is selected as the genotype's mid-point, one-half of the bits need actually be swapped. If the crossover point is selected above one-half of an individual's string length, the swap can take place starting after the mid-point. For example, if there are 72 bits in a individual's string, selecting the crossover point at either position 18 or 54 involves swapping only 18 bits. I will assume that a bit swap involves seven memory operations, namely a read on one word, an AND operation (to mask the bit,) a write of the new word to a temporary location, a read on the second word, an AND operation, and two subsequent writes. I will assume assume that an AND operation requires only one memory operation, namely a masking operation, equivalent to a single write operation. Thus, the number of memory operations required for the population during crossover, $GBLO_{P1}$, is

$$GBLO_{P1} = \left(\frac{1}{4} N\, p_c\, b\, W_T\right) \text{(2 read + 5 write).} \tag{53}$$

A bit inversion during mutation will occur with probability p_m. I will assume this is simply an AND operation, giving the number of memory operations incurred during mutation for the population, $GBLO_{P2}$, as

$$GBLO_{P2} = (N\ p_m\ b\ W_T)\ \text{write}. \tag{54}$$

Now, the total number of operations involved in the parallel section for a single generation, $GBLO_P$, is

$$GBLO_P = \left(\frac{1}{2}\ N\ p_c\ b\ W_T\right)\text{read} + N\ b\ W_T\left(\frac{5}{4}\ p_c + p_m\right)\text{write}, \tag{55}$$

and combining the terms for the serial section and the parallel section,

$$\begin{aligned} GBLO_{total} = &\left(\frac{N^2 + 13N}{4}\right)\text{read}_s + \left(\frac{N^2 + 9N + 4}{4}\right)\text{write}_s + \\ &\left(\frac{N\ b\ p_c\ W_T}{2}\right)\text{read}_p + \left(\frac{5\ N\ b\ p_c\ W_T + 4\ N\ b\ p_m\ W_T}{4}\right)\text{write}_p + \\ &(N-1)\ \text{add}_s + (N+1)\ \text{div}_s, \end{aligned} \tag{56}$$

gives the total number of floating point and memory operations required for a single generation, as executed on a single processor. In Eq. 56, a subscript of s denotes operations that must occur on a single processor (serial operations) and a subscript of p denotes those operations that can be split across multiple processors (parallel operations).

Notice that the number of memory operations required relies heavily on the number of bits used to encode the weights. For example, even when training a small network for the XOR problem, with b=16, N=40, p_c=0.5, p_m=0.01, and W_T=9, $GBLO_{total}$=1970 read +

4091 write + 39 add + 41 div operations. For the larger adder 2 problem with N=480 and W_T=43, $GBLO_{total}$=141,720 read + 268,383 write + 479 add + 481 div. In my implementations I reduce the number of bit operations dramatically by copying and manipulating full bytes or words (two bytes) at a time, rather than individual bits. The crossover term (Eq. 53) is modified to be

$$GBLO_{P1} = \left(\frac{1}{4} N p_c b\right) (2 \text{ read} + 5 \text{ write}) + \left(\frac{1}{4} N p_c W_T\right) (2 \text{ read} + 3 \text{ write}). \tag{57}$$

Here, the first term is the same as in the original equation with the exception of not having the W_T term, because one (and only one) byte or word will typically be split by a crossover point. The second term is a word swap, using only five memory operations--a read, a write to temporary location, another read, and two writes. If this is done, then the number of operations is reduced to

$$GBLO_{total} = \left(\frac{N^2 + 13N}{4}\right) \text{read}_s + \left(\frac{N^2 + 9N + 4}{4}\right) \text{write}_s + \left(\frac{N\, b\, p_c + N\, p_c\, W_T}{2}\right) \text{read}_p + \left(\frac{5\, N\, b\, p_c + 3\, N\, p_c\, W_T + 4\, N\, b\, p_m\, W_T}{4}\right) \text{write}_p + (N-1)\, \text{add}_s + (N+1)\, \text{div}_s. \tag{58}$$

Recalculating the number of memory operations for the XOR network, $GBLO_{total}$=690 read + 1,116 write + 39 add + 41 div. This represents a savings of 1,280 reads and 2,975

writes on a mapping as simple as XOR. For the adder 2 problem, $GBLO_{total}$=66,240 read + 74,523 write + 479 add + 481 div, representing a savings of 75,480 reads and 193,860 writes.

Using the multiplicative factors of M_{SD} and M_{MD}, the equivalent of the number of memory operations required for a backpropagation learning phase (assuming a read and write operation both require the same amount of time) on the XOR problem is ($201 + 54M_{SD} + 89M_{MD}$). For a genetic-based learning algorithm generation, the equivalent of ($1{,}806 + 39M_{SD} + 41M_{MD}$) memory operations would be required.

For the adder 2 problem, an equivalent number of memory operations for the backpropagation learning algorithm is ($1923 + 438M_{SD} + 843M_{MD}$), while for the adder 2 problem, the equivalent of ($140{,}493 + 479M_{SD} + 481M_{MD}$) memory operations would be required.

It should be noted that although the code for the forward computations is the same for each of the algorithms, genetic-based learning requires N times the number of these forward passes that backpropagation requires. Clearly, it can be seen that the backpropagation algorithm is less computationally demanding, and therefore it is desired that the GBL algorithm possess other features which will make it favorable.

6.4 Implementation

As discussed briefly in Section 4.5, when this research work originally began, a backpropagation program was adapted from a book by Pao [7]. Later, for increased performance (decreased real learning time), simulations were developed using hardware and software purchased from Hecht-Nielsen Neurocomputers Corp. However, this was still early in the research, development, and application of neural networks to the inversion of eddy

current data (Chapter 4). As work began on genetic-based learning, there were no commercially available packages (and still are not) or references (which have appeared only recently) for this learning paradigm. As such, all of the software was developed by me, in C++, for use on a variety of machines. The object-oriented characteristics of C++ allowed for a relatively natural mapping of the GBL algorithm into software. The code was developed to be as portable as possible. All attempts were made to eliminate operating system specific calls, such as file operation procedures, memory allocation and deallocation, etc. As an example, initial prototyping and development of the genetic-based learning algorithm and its variants took place on a Compaq 386/25 running under Compaq's DOS 3.31 and using Zortech's C++ compiler. After verification of program operation and results, the source files were ported to an Apollo DN10040 super-mini computer and re-compiled. This machine is based on Apollo's PRISM RISC architecture, and provides much greater performance than the Compaq desktop computer. The environment used here is SystemV UNIX. Few changes to the source were required, most of which related to including the proper header files for the different environment.

Further, the Apollo machine has four CPUs, which greatly aided in developing the parallel algorithm discussed in Section 5.3.4. All of the GBL (and its variants) results presented below were obtained with this parallel algorithm on the Apollo. Because DOS does not support multiple CPUs directly, and because the protocols for communicating between the multiple processors and handling the shared memory segments and semaphores on the Apollo are UNIX specific, this parallel implementation will obviously not run on the Compaq. However, any other multiprocessor, shared-memory system which runs SystemV UNIX should have little trouble running this software.

The backpropagation results presented below were also obtained with software developed by me, from scratch, in C++. Owing to the object-oriented features of C++, I found the

development of backpropagation, as well as many other neural network paradigms (Hopfield, Hamming, Kohonen, etc.), to be much easier than in C. The main reason for this is C++'s reusability of code. In any case, development of the backpropagation learning algorithm took place in a manner similar to that of the GBL code discussed above.

6.5 Comparison of Backpropagation and Genetic-Based Learning Times

Lacking a standardized test suite of mapping problems on which to evaluate learning algorithms, I chose to use a set of binary mappings explored by Rumelhart et al. in Chapter 8 of Parallel Distributed Processing. Binary mappings use only values of one (1) and zero (0) for the inputs and outputs. Each of these mappings will be discussed separately in the following sections. Table 6 lists the mappings along with the number of input neurodes, hidden neurodes, and output neurodes used for each test case. Each of these binary mappings is used to explore the number of iterations (for backpropagation) or generations (for genetic-based learning) required to obtain an acceptable solution. The backpropagation algorithm utilizes a momentum term, and is abbreviated as BPM in the tables. Several different versions of genetic-based learning are explored in order to show the merits of employing the enhancements discussed in Section 5.3. The first algorithm uses standard genetic algorithm operators, and is abbreviated GBL. The second adds a window to induce selective pressure as well as using an elitist model, thus abbreviated as GBLWE. The last two versions employ a window and an elitist model, but also include two different versions of an adaptive mutation operator (see Section 5.3.3). One uses a direct Hamming distance measurement, as used by Whitley and Hanson [37], while the other utilizes a power weighted Hamming distance

measurement, as explored by myself. These two different versions are appropriately labeled in the tables.

Table 6. Binary mappings used in comparing genetic-based learning with backpropagation learning

Mapping	Number of inputs neurodes	Number of hidden neurodes	Number of output neurodes
Parity M	M	M	1
Symmetry M	M	2	1
Encoder M	M	$\log_2 M$	M
Adder 2	4	5	3

6.5.1 Selection of the mutation probabilities

One major point concerning the adaptive mutation operators not discussed previously is how the selection of the upper and lower bounds is made. After much exploration, I have identified two methods of determing these bounds that appear to be effective. By recording numerous trials and identifying those mutation probabilities which appeared to be more effective than others, it was determined that the average probability of mutation selected during a complete simulation, using the adaptive operators, remained between 1/8 and 1/12 of the difference between the upper and lower bounds. Therefore, both methods of determining the upper bound are based on this result. For simplicity, I leave the lower bound at zero.

The first method of selecting the upper bound of the mutation probability is based only on the number of bits used to encode a weight in the network. This implies a rate of mutation which applies directly to the length of a weight, not the length of a genotype. As the size of the

network grows, and therefore the length of a genotype grows, the number of bits which will likely be mutated in a genotype grows, while the number of bits likely to be mutated in each weight remains constant. The equation developed for determining the upper bound is given as

$$p_{mu} = \sqrt{\frac{A\,x}{b}}, \tag{59}$$

where A is a multiplicative factor, usually between eight and 12 (as discussed above), x is the desired number of mutations per weight, and b is the number bits used to encode a weight.

The second method is based on the length of the genotype, which is determined by the number of weights in the network and the number of bits used to encode a weight. In this case

$$p_{mu} = \sqrt{\frac{A\,x}{b\,W_T}}, \tag{60}$$

where W_T is the total number of weights in the network. For the simulations run in performing the comparisons below, I used A=12 and x=0.25. The value for p_{mu} appears in column six, when appropriate. For the GBL and GBLWE algorithms not employing an adaptive mutation operator, column six contains the value of a constant mutation probability. Again, there are two different values, one based on Eq. 59 and the other on Eq. 60.

6.5.2 Details of the learning comparisons

All of the results presented in the following sections were acquired through 25 trials, in which each trial uses a different random seed for initializing the weights of the network while

holding all of the other parameters constant. For all learning algorithms, a fully connected, feed-forward network with a single hidden layer was used. Further, each neurode used a sigmoidal activation function with a lower bound of zero, an upper bound of one, and a slope of one (see Section 3.6).

For the backpropagation simulations, the learning rate, α, is 0.5 or 0.7 and the momentum rate, η, is 0.7 or 0.9, as indicated in column six. Also, a batch size (see Section 3.5) equal to the number of input-output pairs in the training file was used in order to make the comparison as fair as possible.

For the genetic-based simulations, the population size, N, is 10 times the number of input-output pairs in the training set times the number of output neurodes. 16 bits are used to encode each weight in a network, while the range of weights is ±64.0. The probability of crossover, p_c, is 0.5 Several different values for the mutation rate, p_m or p_{mu}, were used and are given in column six where appropriate.

The MSE goal for each of the mappings was set to 0.0001, which is a very low value for binary mappings. I felt this would stringently test each of the algorithms in its ability to generate a correct mappings.

Column five of each of the results tables shows the number of unconverged trials. By unconverged, I mean that if the mapping problem was not solved (the MSE was not reduced to the goal level or below) by the time a predetermined maximum number iterations or generations had been reached, then the trial was stopped. The maximum number of iterations or generations varies for the BPM and GBL algorithms and for the problem under investigation. These maximum values are given in each of the following sections.

Finally, columns three and four give the average number of iterations and standard deviation of the number of iterations, respectively, required in obtaining a solution. An iteration, as used here, is defined to be a pass through the entire training set culminating in a

single update of the weights at the end of the pass. These values are computed only for those trials which actually converged. Therefore, if only 20 of 25 trials converge for a particular algorithm and problem, then the average and standard deviation are computed with the 20 converged trials.

6.5.3 Parity M

Parity, as used in the computer field, refers to an error checking procedure in which one or several additional bits are appended to a vector of data bits. In these test cases, parity M means that for M bits in the input string, a single parity bit would be appended to the vector in order to provide error checking. If the number of bits in the input vector that are ON (have a value of one) is odd, then the parity bit is a one, otherwise it is a zero. The goal of the network for these mappings is to produce the correct parity bit at the output layer given the input vector. Tables 7 and 8 give the input and output values for the parity 2 and parity 3 mappings, respectively. Notice that the parity 2 mapping is identical to the XOR problem described earlier.

Table 7. The parity 2 mapping

Input x_1	Input x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 8. The parity 3 mapping

Input x_1	Input x_2	Input x_3	Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tables 9 and 10 show the results of training networks to perform the parity 2 and parity 3 problems, respectively, for different learning algorithms and different parameter settings. For parity 2, the maximum number of iterations allowed for the BPM simulations was 10,000, while the maximum number of generations allowed for each of the GBL simulations was 1,000. For parity 3, the maximum values were 15,000 for BPM and 5,000 for GBL.

For the parity 2 problem, it is clear that all of the GBL algorithms have performed much better than backpropagation, meaning that many fewer generations were required for the GBL algorithms. Further, the learning enhancements have improved the performance of the GBL algorithm somewhat. Notice that several of the trials for each of the BPM simulations remained unconverged, suggesting that the algorithm became trapped in a local minima. All of the GBL trials converged.

For the parity 3 problem, it is apparent that the GBL algorithms without the adaptive mutation operator have not performed very well. This suggests that this operator helps significantly in decreasing the number of generations required to reach a solution. It is believed that the GBL and GBLWE algorithms would eventually solve the parity 3 problem

consistently, given more generations. In other words, these algorithms have not necessarily become trapped in a local minima, they merely lack genetic diversity in the populations and would require more time, relying on a constant mutation rate, to explore new regions of the search space. A dramatic difference is seen when the adaptive mutation operators are employed. All of the trials converged. Further, the average number of generations required dropped significantly when the second method for selecting the upper bound of the mutation probability was employed.

Table 9. Comparison of learning algorithms on the parity 2 (XOR) mapping problem

Learning algorithm	Mutation property	Average iterations	Std. dev. of iterations	Number un-converged	Parameter settings
BP	Not Applicable	2112.14	445.31	3	α=0.7 η=0.9
BP	Not Applicable	8346.33	435.06	4	α=0.5 η=0.7
GBL	Constant	54.92	53.28	0	p_m=0.433013
GBL	Constant	66.20	59.82	0	p_m=0.408248
GBLWE	Constant	57.60	64.11	0	p_m=0.433013
GBLWE	Constant	44.80	39.78	0	p_m=0.408248
GBLWE	Whitley	14.44	18.16	0	p_{mu}=0.433013
GBLWE	Power	11.96	9.78	0	p_{mu}=0.433013
GBLWE	Whitley	12.04	8.47	0	p_{mu}=0.408248
GBLWE	Power	9.68	6.59	0	p_{mu}=0.408248

Table 10. Comparison of learning algorithms on the parity 3 mapping problem

Learning algorithm	Mutation property	Average iterations	Std. dev. of iterations	Number un-converged	Parameter settings
BP	Not Applicable	1424.72	438.77	0	α=0.7 η=0.9
BP	Not Applicable	5746.92	777.79	0	α=0.5 η=0.7
GBL	Constant	2479.00	1671.95	10	p_m=0.433013
GBL	Constant	2484.53	1269.99	10	p_m=0.306186
GBLWE	Constant	1913.64	1192.35	11	p_m=0.433013
GBLWE	Constant	2716.18	1256.41	8	p_m=0.306186
GBLWE	Whitley	151.16	174.68	0	p_{mu}=0.433013
GBLWE	Power	83.76	99.85	0	p_{mu}=0.433013
GBLWE	Whitley	36.40	18.90	0	p_{mu}=0.306186
GBLWE	Power	53.36	83.73	0	p_{mu}=0.306186

It is my belief that the large difference in number of unconverged trials for the GBL and GBLWE algorithms between the parity 2 and parity 3 problems results from an increased search space. The parity 2 problem is quite small and is much easier to find a solution for, even randomly, perhaps, than the parity 3 problem. Increasing the size of the search space requires more of the genetic algorithm properties to function effective in finding an "optimal" solution. This means maintaining high genetic diversity, whether through an increased population size or the adaptive mutation operator. This point will be further illustrated in the following sections.

6.5.4 Symmetry M

The symmetry problem involves determing if the input data vector is symmetric with respect to its center. Table 11 lists the input and output values for symmetry 4. Clearly, the symmetry mapping is only applicable for input vectors which have an even number of bits.

Table 12 shows the results of training networks to perform the symmetry 4 problem. For the BPM simulations, the maximum number of iterations allowed was 50,000, while the maximum number of generations allowed for each of the GBL simulations was 15,000.

Table 11. The symmetry 4 mapping

Input x_1	Input x_2	Input x_3	Input x_4	Output
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 12. Comparison of learning algorithms on the symmetry 4 mapping problem

Learning algorithm	Mutation property	Average iterations	Std. dev. of iterations	Number un-converged	Parameter settings
BP	Not Applicable	654.79	44.32	11	α=0.7 η=0.9
BP	Not Applicable	2746.00	69.68	12	α=0.5 η=0.7
GBL	Constant	4628.00	0.00	24	p_m=0.433013
GBL	Constant	------	------	25	p_m=0.339683
GBLWE	Constant	------	------	25	p_m=0.433013
GBLWE	Constant	------	------	25	p_m=0.339683
GBLWE	Whitley	342.96	667.09	0	p_{mu}=0.433013
GBLWE	Power	402.96	382.62	0	p_{mu}=0.433013
GBLWE	Whitley	355.00	541.97	0	p_{mu}=0.339683
GBLWE	Power	222.80	207.53	0	p_{mu}=0.339683

Here, it can be seen that when backpropagation was able to solve the symmetry 4 problem (about half of the time), it did so relatively quickly using high values of α and η. However, it is clear from the resulting MSE of those trials for BPM that were left unconverged, that the algorithm had become trapped in a local minima. This means that the MSE was relatively large, not near the goal.

On the other hand, all of the GBL algorithms employing the adaptive mutation operator solved the problem every time, and did so about twice as fast as when BPM solved it. Again, it is noted that the fastest GBL adaptive method uses the second method of selecting an upper bound on the mutation probability.

6.5.5 Encoder M

In the encoder problem, the inputs and output of the mapping are the same, with the restriction that only one bit of the M inputs is on. The job of the network is to encode the information using as few neurodes as possible. Theoretically, given M binary inputs, the information can be encoded with $\log_2 M$ hidden neurodes. This requires the network to develop an internal binary coding of the input vector. Table 13 gives the encoder 4 mapping.

Table 13. The encoder 4 mapping

Input x_1	Input x_2	Input x_3	Input x_4	Output y_1	Output y_2	Output y_3	Output y_4
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
1	0	0	0	1	0	0	0

Table 14 shows the results of training networks to perform the encoder 4 problem. For the BPM simulations, the maximum number of iterations allowed was 50,000, while the maximum number of generations allowed for each of the GBL simulations was 15,000.

The results using the GBL algorithms with an adaptive mutation operator show a significant improvement (several orders of magnitude) over that of BPM. Again, however, the GBL algorithms without this adaptive method have failed, due most probably to a severe lack of genetic diversity.

Table 14. Comparison of learning algorithms on the encoder 4 mapping problem

Learning algorithm	Mutation property	Average iterations	Std. dev. of iterations	Number un-converged	Parameter settings
BP	Not Applicable	8566.88	1777.22	0	α=0.7 η=0.9
BP	Not Applicable	34550.70	3412.15	2	α=0.5 η=0.7
GBL	Constant	------	------	25	p_m=0.433013
GBL	Constant	------	------	25	p_m=0.261116
GBLWE	Constant	2822.33	1546.82	22	p_m=0.433013
GBLWE	Constant	1937.50	1053.50	23	p_m=0.261116
GBLWE	Whitley	269.76	520.66	0	p_{mu}=0.433013
GBLWE	Power	161.08	278.45	0	p_{mu}=0.433013
GBLWE	Whitley	94.84	183.16	0	p_{mu}=0.261116
GBLWE	Power	39.52	13.94	0	p_{mu}=0.261116

6.5.6 Adder 2

The adder 2 mapping is designed to simulate a two-bit adder. That is, given two two-bit numbers, the network is to produce the correct three-bit output which represents the addition of the two two-bit numbers. Three output bits are required, one representing the carry bit. Table 15 lists the input and output values for the adder 2 problem. Output y_1 is the most significant digit (MSD) or carry bit and output y_3 is the least significant digit (LSD).

Table 15. The adder 2 mapping

Input x_1	Input x_2	Input x_3	Input x_4	(MSD) Output y_1	Output y_2	(LSD) Output y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Table 16 shows the results of training networks to perform the adder 2 problem. For the BPM simulations, the maximum number of iterations allowed was 100,000, while the maximum number of generations allowed for each of the GBL simulations was 25,000.

Table 16. Comparison of learning algorithms on the adder 2 mapping problem

Learning algorithm	Mutation property	Average iterations	Std. dev. of iterations	Number un-converged	Parameter settings
BP	Not Applicable	4432.77	3814.20	3	α=0.7 η=0.9
BP	Not Applicable	13194.20	13082.70	1	α=0.5 η=0.7
GBL	Constant	------	------	25	p_m=0.433013
GBL	Constant	------	------	25	p_m=0.186772
GBLWE	Constant	------	------	25	p_m=0.433013
GBLWE	Constant	------	------	25	p_m=0.186772
GBLWE	Whitley	8436.45	8060.89	14	p_{mu}=0.186772
GBLWE	Power	3325.91	5224.04	14	p_{mu}=0.186772

6.6 Conclusions and Discussion

After running numerous simulations on various problems, including some which are included in this Chapter, I have found the performance of GBL, especially the variants incorporating the learning enhancements, to be more than acceptable. Often, GBL outperforms backpropagation in time required for learning a particular mapping problem and in the quality of the solution. When the GBL algorithm is able to solve a problem, it usually produces MSE values which are much lower than that produced by backpropagation. This is because backpropagation is generally quite slow at fine tuning the weights in a network once a good area of the search space has been located. In order to increase the rate at which this fine tuning progresses, it is necessary to increase the learning parameters. However, doing so may cause

the network to work itself out of the good region--essentially move back up the gradient descent, not down it!

However, the trouble the GBL algorithms show in solving the adder 2 problem is puzzling and of concern to me. I believe the main cause of this trouble relates to the multiple output neurode network involved and the size of the network needed for solving this problem. The increased dimensionality of the network creates a much larger search space for the algorithm, thus requiring more effort in finding good weights. I have three suggestions for improving GBL's performance on networks of this type. First, use a larger population, thus enabling more of the search space to be explored simultaneously. Second, split the problem into several pieces, each one of which consists of only a single neurode. This improves the mapping by producing a four to one mapping instead of a four to three mapping. Third, train the network in stages, say one hidden neurode at a time. This method is presented in Chapter 7 with the introduction of Dynamic Node Creation.

Also of interest is the difference between the two adaptive mutation operators explored in this chapter, the straight Hamming distance measure as used by Whitley and Hanson [37] and the power Hamming distance measure as explored by me. In all cases, with the single exception of the parity 3 mapping, the power Hamming distance measure produced acceptable results in fewer iterations and with a lower standard deviation. This indicates to me that this method of monitoring genetic diversity is better than using a straight Hamming distance measure which does take into account the position in which bit differences occur.

Because the GBL algorithms are naturally and easily parallelized, I believe this is one of the most important advantages which GBL provides over other single-threaded, iterative training procedures, such as backpropagation. This, coupled with the fact that many fewer iterations of the algorithm are often required makes GBL attractive for use in solving problems more accurately and more quickly.

Finally, while there is still much work to be done on using genetic algorithms for optimizing neural networks, I believe that progress will be made quickly once the allure of this powerful paradigm is realized.

7 NEURAL NETWORK ARCHITECTURE CONSIDERATIONS

7.1 Overview

One major aspect of artificial neural network performance which has been overlooked until this point is that of architectural considerations. I interpret neural network architecture as relating to two major issues. The first is the connectivity pattern of the neurodes in a network. Previous chapters have considered only fully connected feed-forward networks, in which each neurode in a layer is connected to every neurode only in the layer directly above it. However, many other patterns of connectivity have been investigated, some of which are used in paradigms other than feed-forward networks, some of which remain in the realm of feed-forward networks. Examples of connectivities in paradigms other than feed-forward include the Hopfield model [6], the Adaptive Resonance Theory (ART) models [6], and Bidirectional Associative Memories (BAMs) [6]. Examples of connectivities other than fully connected within the feed-forward domain include random interconnections, recurrent and/or intralayer interconnections, connections which skip layers, receptive field connections [43], or any combination. Although this issue of neural network architecture is certainly interesting and under active research, I will not be concerned with the matter of patterns of connectivity throughout the remainder of this chapter. My investigations will remain based on the concept of fully connected feed-forward networks (see Fig. 5).

The second major area of neural network architecture is that of network size, or dimension. Network size relates to the number of neurodes and number of layers required to

obtain a good, or "optimal", solution to the problem being solved. Because I will only be discussing feed-forward networks, when referring to the number of neurodes required to solve a particular mapping problem, I am referring to the number of neurodes required in the hidden layer of a network. This assumes that the mapping problem appropriately describes the number of input and output neurodes required for the network, so that these two layers are fixed for any given problem. Further, in order to limit the scope of the material involved, I will consider only networks with a single hidden layer. While it is known that additional layers can provide additional levels of abstraction and feature detection [7,49], all of the problems used as examples in this chapter are known to be solvable with a single hidden layer. Ideally, the research presented in this chapter could be expanded to explore how additional layers of hidden neurodes can be used to help solve other mapping problems.

The remainder of this chapter discusses aspects of the number of neurodes required in solving certain mapping problems, namely those used in Chapter 6 for comparing backpropagation and genetic-based learning. Section 7.2 provides the motivation for selecting a minimal, or “optimal”, number of neurodes in the hidden layer of a neural network. Section 7.3 gives some background on research and results concerning the determination of the number of hidden neurodes required for particular mapping problems. Section 7.4 discusses an abstract notion pertaining to feature detection in neural networks, setting up a discussion of how this idea can be used to alter a method of dynamically configuring neural networks which can require less computation than other methods. Section 7.5 gives a detailed description of Dynamic Node Creation (DNC), a method of dynamically inserting additional hidden neurodes into neural networks at "appropriate" times in order to increase the network's ability to solve the problem under investigation. Section 7.6 will discuss some issues of DNC which I believe to be relevant to feature extraction and detection, but which have not been considered in other publications. This section also presents results of simulations performed by me using the

popular XOR mapping problem which shows the relevance of the feature detection hypotheses. Conclusions and discussion are contained in Section 7.7.

7.2 Motivation

What role does the architecture of a neural network play in solving mapping problems? On an abstract level, the hidden layers of feed-forward networks provide increasingly higher levels of feature extraction. Hidden layers are able to extract certain information from their inputs and essentially reorganize that information into a form which is more suitable for producing desired output responses. For example, it was discussed in Chapter 2 that a perceptron without a hidden layer of neurodes was not able to solve a problem as simple as parity 2 (XOR). It was also noted that the network's inability to solve this problem was related the fact that parity 2 is not a linearly separable mapping, and that any mapping which possesses this property will present a similar problem for single layer perceptron models.

Several issues related to the addition of a hidden layer to solve XOR were not discussed, including how the hidden layer reorganizes the input information into more useful information, why and how this feature detection or extraction works, and could a network with a number of hidden neurodes other than two solve XOR? The first two issues will be considered in a separate section (see Section 7.4), while the latter is discussed briefly here.

Although two hidden neurodes is the minimum number required for solving XOR, certainly more neurodes could be used. A single neurode in the hidden layer would be equivalent to a single layer perceptron, so that the only function which could be performed in moving from the single hidden neurode to the output neurode would be to attenuate or amplify

the output of the hidden neurode. This would serve only to invert or accentuate the decision of the hidden neurode. Therefore, no less than two hidden neurodes can be used.

For the XOR problem, what happens when the number of hidden neurodes is increased beyond two, the "optimal" number for XOR? First, by using more hidden neurodes than necessary, more computation than necessary is used in evaluating new cases, as well as during learning. For small problems such as XOR, adding a few more neurodes will not present much of an increase in computation time in either learning or forward evaluation. However, when solving much larger problems involving possibly hundreds or thousands of neurodes, the computational requirements can certainly become large, and adding extra neurodes serves to extend the time required for evaluation and training. Depending upon the number of extra neurodes used, the added time could be significant. Therefore, to decrease learning time and evaluation of new cases after learning, a minimal (optimal) set of neurodes is preferable. Second, additional neurodes require additional memory. Again, for small networks and relatively simple mapping problems, this will not be a hindrance; larger networks and mappings can introduce problems in memory usage, especially if training is performed on a small system, such as a desktop computer. Third, and perhaps most importantly, a tradeoff is made between generalization and memorization when selecting the number of neurodes used in solving a problem.

This tradeoff is best illustrated through an example. Consider the set of data points, marked X and O, in Fig. 21. The x-axis is the input to the function and the y-axis represents the result of applying a function f to the x-axis input data. The points lie along a curve which is easily recognized simply by glancing at it. The job of the network is to discover this general relationship between the x data points presented during training and the curve which represents the function that describes the mapping from the input x to the output y=f(x). The points marked with an X are included in the training set, while the points marked with an O comprise

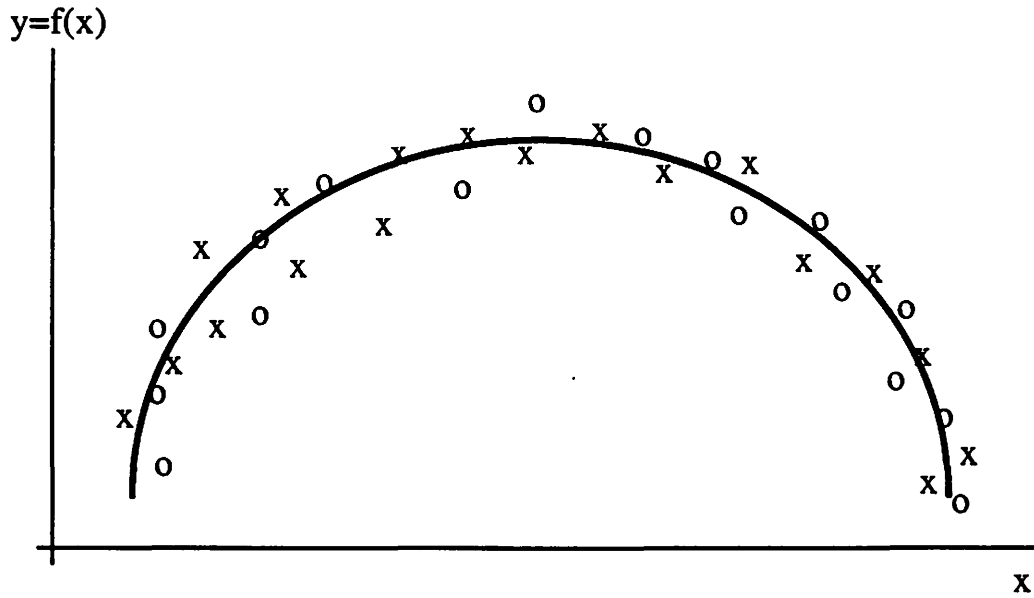


Figure 21a. Result of a properly trained network with the ability to generalize to novel input data.

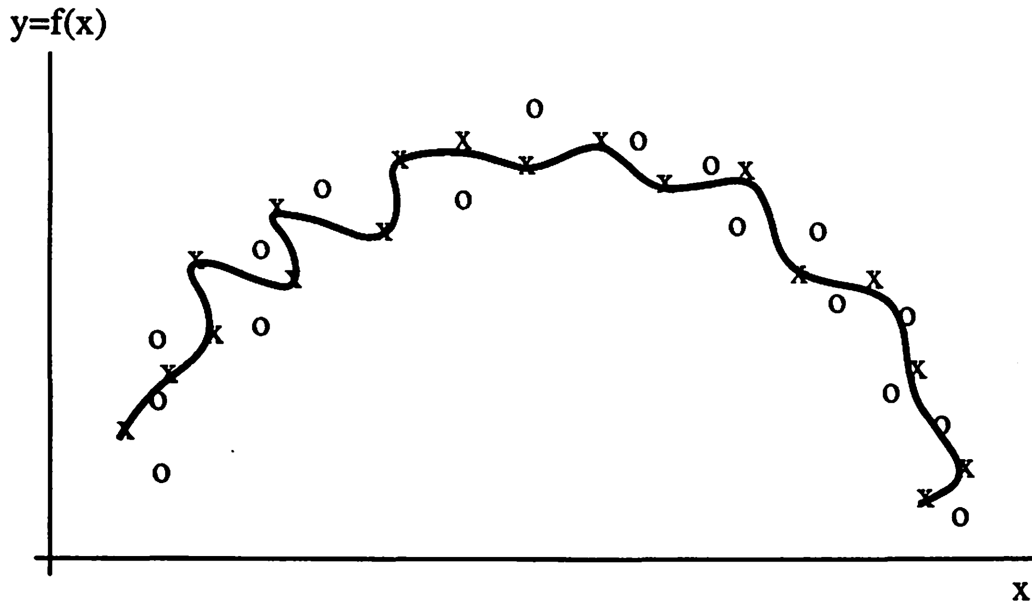


Figure 21b. Result of an improperly trained network which has memorized the training data.

the testing set, seen by the network only after training. Figure 21a shows the result of properly training the network so that it will be able to correctly predict outputs for input values not previously seen during training (novel input data). The solid line is the network's prediction, after training, for all possible input values--the function, or mapping, which the network has discovered during training. This ability to produce correct responses for novel data is known as generalization. On the other hand, Fig. 21b shows what can happen when a network is improperly trained. Here, the network has overfit to, or memorized, the training data. It has become so specialized that it properly recognizes only that data presented during training. As a consequence, the network's ability to generalize has been sacrificed. This case of memorization can occur for several reasons, just one of which is using too many hidden neurodes.

Therefore, it is desirable to have the ability to obtain, or know, the optimal number of neurodes required for any particular mapping problem. The method of training a correctly sized neural network still remains much of an art, rather than a science. Thus, any method which can be employed to usefully aid a neural network designer in generating the best network possible is welcome.

7.3 Background

As it stands, the problem of selecting the correct number of neurodes (for the hidden layer) is a least as important as, and often more difficult than, selecting good learning algorithms and the parameters used during learning. No one has developed a theory describing the exact number of neurodes and the activation function to be used by the neurodes in order to solve an arbitrary mapping problem. Numerous studies have been conducted in order to set

upper and lower bounds on the number of neurodes needed when using certain activation functions or to give "rules of thumb" or guidelines [44] to the number of neurodes needed. While these rules and guidelines may hold for certain classes of mapping problems, activation functions, and network connectivities, they are not general enough to be of use in all situations. This limited usability in real world situations makes these methods questionable at best.

Robert Hecht-Nielsen [45] has applied an existence theorem provided by Kolmogorov [46] in solving the 13th problem of Hilbert [47], which states, briefly, that any continuous function can be mapped exactly with a neural network consisting of an input layer, an output layer, and one hidden layer of (2n+1) neurodes (where n is the number of input neurodes), given that the appropriate activation functions are used in the hidden and output layers. However, this is merely an existence theorem in that it provides no method of determining the activation functions to be used.

7.4 Feature Detection

One of the dark spots that remains in using neural networks to solve problems is the network's inability to explain how a conclusion is reached. Using expert systems, it is a simple, and often necessary, step to include an audit trail feature which can detail the rules fired and facts used in making a decision. This lack of an explanation facility in neurocomputing has prompted researchers [48] to begin looking for ways in which rules might be built from trained networks. One interesting aspect of this problem is feature detection.

Abstractly speaking, it is useful to consider neural networks as detecting, or extracting, features from the input vector in order to develop a mapping and produce correct output responses. In fact, Lippman [49] illustrates how such feature detection is done in networks

with various numbers of layers. As an example, consider the XOR problem. It has been noted repeatedly than a perceptron without a hidden layer can not solve this problem. This is because the perceptron, for a two-input vector, can form only a single linear decision boundary. Such a boundary is illustrated in Fig. 3. If a hidden layer of one neurode is added to the perceptron model, this hidden neurode can similarly create only a single linear decision boundary, and with only a single input, the output neurode can only amplify or invert the hidden neurode's decision. However, adding a second hidden neurode (see Fig. 4) allows for the creation of two linear decision boundaries, as shown in Fig. 22. Each of the decision boundaries isolates one of the XOR points which should be classified as a one. That is, each hidden neurode responds positively, or fires, only when the feature it has been trained to respond to is present in the input vector. These two hidden neurodes perform the exclusive portion of the XOR. Now, the output neurode can combine the information from each of the hidden neurodes, performing an OR operation, and correctly classify the input data. Adding more neurodes to the single hidden layer provides the ability for the network to create more complex regions. In the XOR example, this would be "overkill" as only the two linear decision boundaries are required.

This feature detection ability of neural networks is a powerful tool in analyzing the information stored in the distributed representation of the network. However, if too many hidden neurodes are used, then the usefulness of this feature detection analysis can be negated. For example, if four hidden neurodes are used for the XOR problem, then the network can be trained so that each of the hidden neurodes would respond to only one of the input vectors. Such a network is shown in Fig. 23. In this figure, neurode three responds only to the input (0,0), neurode four to (0,1), neurode five to (1,0), and neurode six to (1,1). In this case, all feature detection characteristics have been lost because the network has memorized each of the

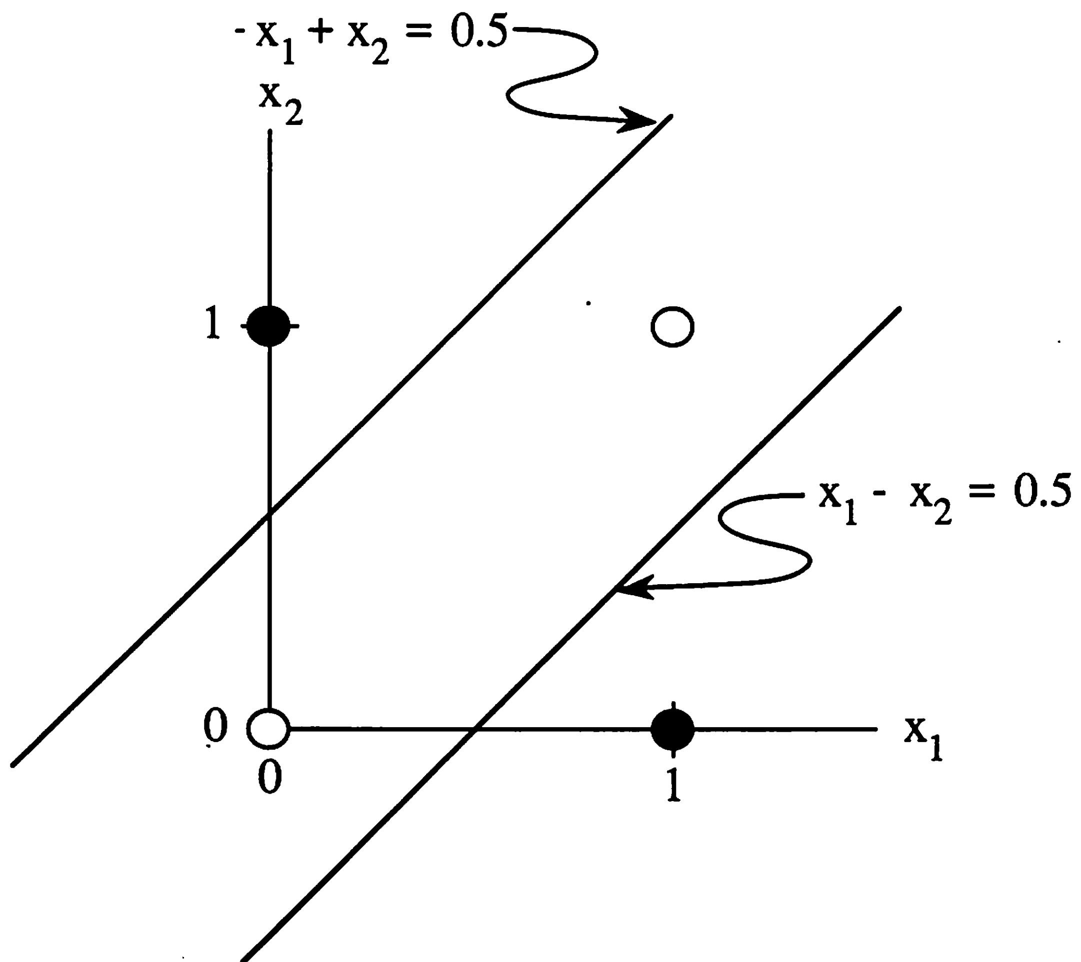


Figure 22. Two linear decision boundaries formed by a perceptron model with two hidden neurodes for the XOR problem.

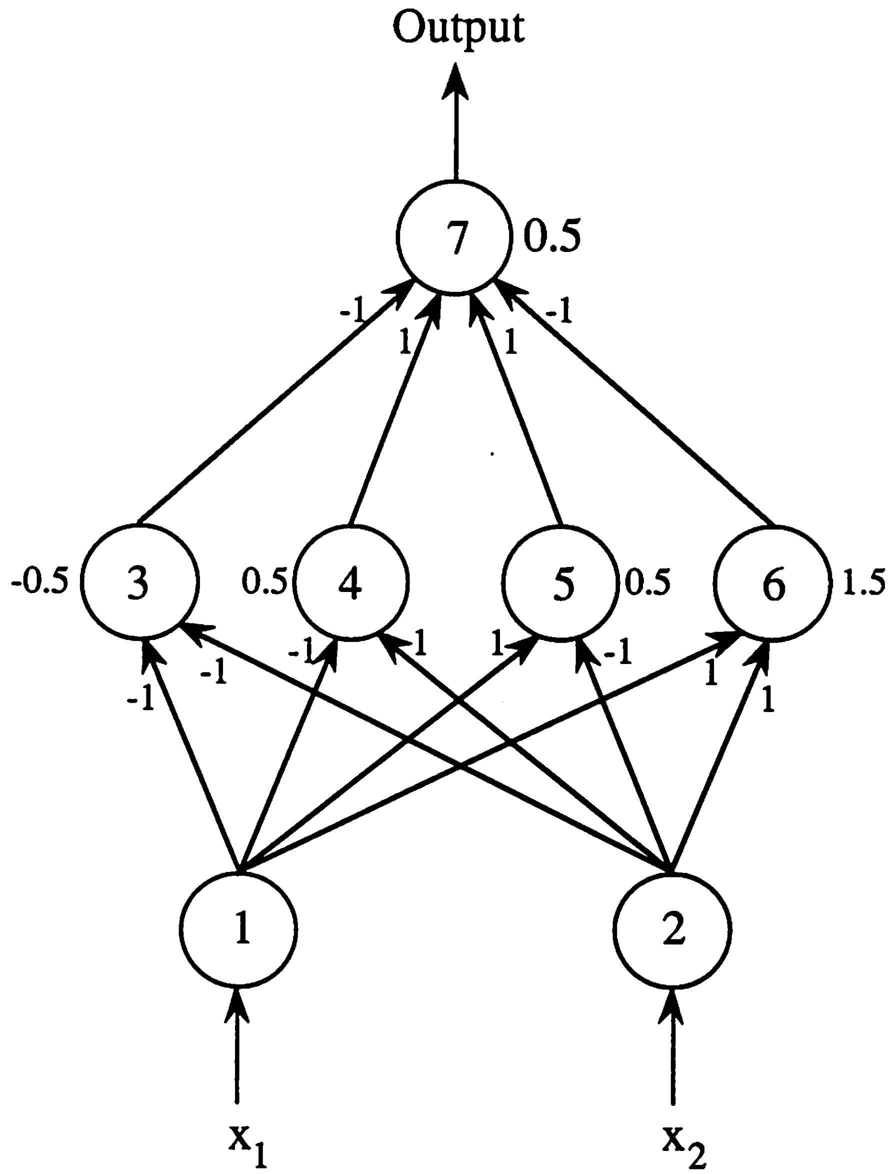


Figure 23. A four hidden neurode perceptron model for the XOR problem.

input vectors. For a more complex problem, generalization of novel input data is forfeited when this happens.

7.5 Dynamic Neurode Creation

Lacking a rigorous model for determining a priori the number of hidden neurodes required for solving a problem, Dynamic Neurode Creation (DNC), a technique originally published by Timur Ash [11], is a heuristic approach of dynamically configuring neural networks. It attempts to aid a network during learning by providing more degrees of freedom through the addition of hidden neurodes while the search is in progress.

In doing so, one major difficulty must be solved--when to add another neurode. The time at which additional neurodes are added can play a large role in determining how well the system performs in solving the problem in an optimal, or near-optimal, manner. When training a multi-layer feed-forward network using an optimal number of neurodes with backpropagation, it is typical to see a period in which the MSE of the network decreases quite rapidly and then levels off. This leveling off period denotes the period of training when fine tuning of the system occurs. The network has already discovered a mapping which is close to that desired, but which still requires minor adjustments to the weights in order to more closely realize the appropriate mapping. When training networks which have fewer than the optimal number of neurodes, this same pattern is typical. However, the leveling off period corresponds to one in which the network has performed as well as possible, given the number neurodes it has. In order for the MSE to decrease substantially from its current level, more hidden neurodes must added. This is known because the MSE of the network remains quite

high. Ash detects this leveling off period and uses this information to guide the system in determining an appropriate time in which to add another neurode.

Two considerations are required for detecting this leveling off period. The first is a way of detecting when the error rate has actually leveled off, representing a substantial lack of progression in learning further. In doing so, three reference points are needed. The first, t_0, is the time step, or iteration, at which the last hidden neurode was added. The second, t, is the current iteration. The third, (t-w), is w iterations previous to the current iteration. w defines a window over which the MSE is monitored. Using these three reference points, the ratio of the drop in MSE over the last w iterations to the MSE when the last neurode was added can be computed as

$$\frac{|MSE_t - MSE_{t-w}|}{MSE_{t_0}}. \qquad (61)$$

When this ratio falls below a preset level, which Ash calls the trigger slope, another hidden neurode should be added, if and only if

$$t - w \geq t_0. \qquad (62)$$

Equation 62 ensures that all of the MSE terms refer to the same network architecture.

Figure 24 shows the relationship of these terms on a representative, monotonically decreasing MSE curve. Once a new hidden neurode is added to the network, training continues on the entire network, using the previously trained neurode weights (for the old neurodes) and randomized weights for the new neurode. No more hidden neurodes are added when it is determined that the network has solved the mapping problem satisfactorily, as

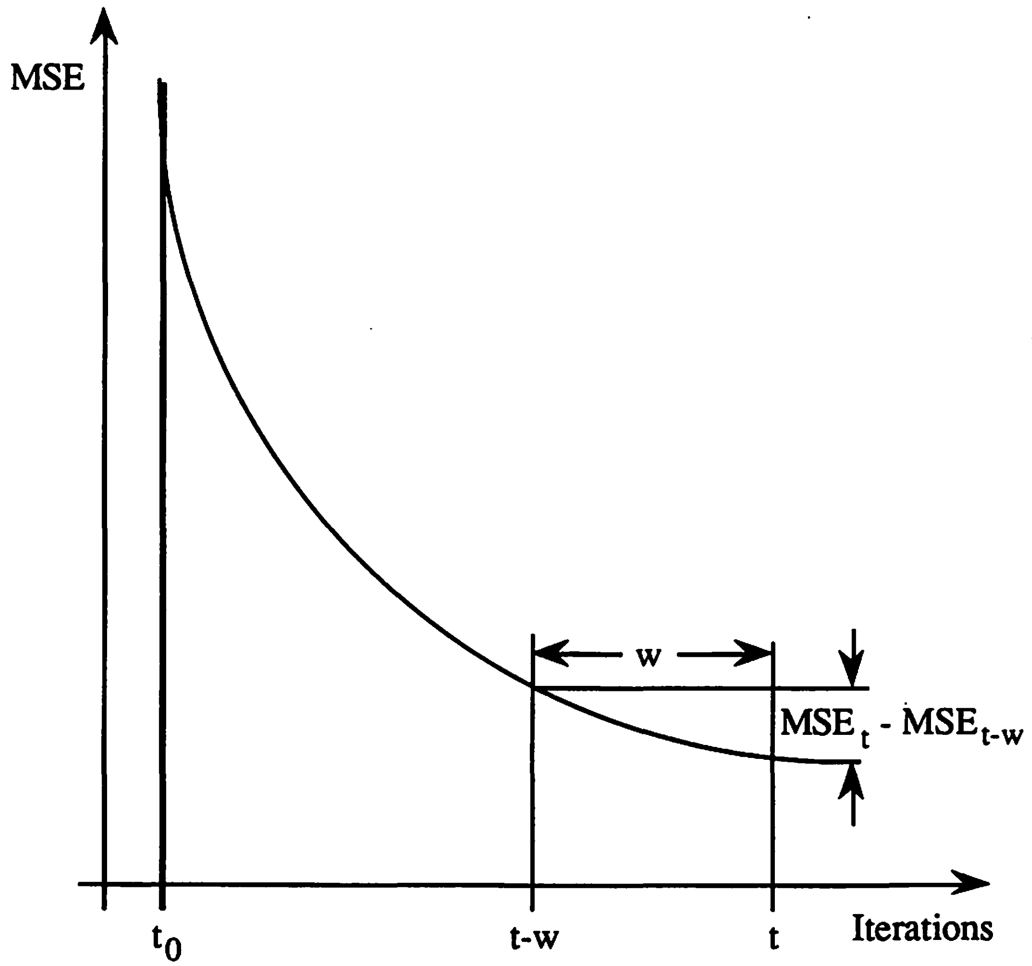


Figure 24. Detection of the leveling off period in a neural network with too few hidden neurodes to solve the problem at hand.

required by the implementor. Both Ash and myself have used this method of searching for optimal network architectures successfully. All of the results obtained in the following sections use this method of determining when the addition of a new hidden neurode is to take place.

One interesting aspect which Ash brings to light is that the computational requirements for training an optimally sized neural network using DNC is often much less than starting with the optimal network architecture. The reason for this will be discussed in detail in the following section.

7.6 Feature Detection Using Dynamic Neurode Creation

In discovering optimal or near-optimal solutions with DNC, the learning algorithm typically spends much less time in training the network once it has reached the optimal size than it would if it started with an optimal network architecture. Ash points out that this implies training which has occurred with fewer neurodes than optimal helps the learning algorithm in finding good solutions. While this seems to be true, I believe there is a more fundamental reason, one which bears further examination and can help to reduce the amount of computation required in reaching the optimal architecture, as well as once the optimal size is reached.

Returning to the idea of feature detection and extraction, I propose that in reaching an optimal network architecture, hidden neurodes can be trained, individually, to respond to different features in the input vectors. By this I mean that when a new hidden neurode is added, the old hidden neurodes' weights are "frozen"--no more learning occurs on these weights. However, learning still occurs on the weights connecting the old hidden neurodes to the output layer neurodes. This is necessary because when a new neurode is added to the network, the old hidden neurodes may have been properly trained as detectors for certain

features in the input vector, but the output neurodes still need to be able to combine these features appropriately in order to produce correct output responses. If this proposition is correct, then at each stage of neurode addition very little additional computation is required--only one more weight at each of the output neurodes is added into the learning algorithm.

The following sections will show how this proposition holds, and what effect it has on finding optimal or near-optimal solutions in neural networks. My investigations into this method of dynamically configuring networks originated due to long training times for large networks and mapping problems using genetic-based learning. I felt that if, at each stage of training a subportion of a network, good feature detection mechanisms were apparent, then networks could be trained in stages, each of which is only slightly larger than the previous stage. This would allow for possibly faster training of the entire network, and perhaps a higher percentage of converged trials.

7.6.1 XOR feature detection

The first example to be considered is the popular XOR (parity 2) problem. Figure 4 is just one of many multi-layer perceptron networks which is capable of solving this problem. One solution for a network which incorporates a sigmoidal activation function is similar to this, but due to the variety of ways in which the XOR problem may be solved, there exist many networks using nonlinear activation functions with the ability to solve this problem. The next three subsections all contain networks which solve the XOR problem in a similar manner. However, each of these networks is obtained in a different way.

7.6.1.1 Normal backpropagation learning for the XOR problem The first method of developing a neural network to solve the XOR problem utilizes standard backpropagation with a momentum term (see Section 3.5) and two hidden neurodes at the beginning of training with all neurodes using a sigmoidal activation function. This is the method that would normally be used in training a network to perform a particular mapping--select a learning paradigm, the network architecture, values for the learning parameters, and then commence training. Figure 25 shows the resulting network and Table 17 gives the output response for each non-input neurode in the network for each of the four input-output training pairs.

Table 17. Output values for each training input-output pair and every non-input neurode in Fig. 25

Input x_1	Input x_2	Output for neurode #3	Output for neurode #4	Output for neurode #5
0	0	0.972	0.039	0.010
0	1	1.000	0.942	0.991
1	0	0.031	0.000	0.989
1	1	0.973	0.031	0.009

In this network, neurode number three has been trained to fire to any pattern except (1,0), therefore singling out this pattern as being different, or unique--in essence, detecting this feature in the input vector. Note that the output neurode inverts this neurode's signal, thereby recognizing in an excitatory manner when the pattern (1,0) is present at the input neurodes. Likewise, neurode number four has been trained to fire only when the pattern (0,1) is present

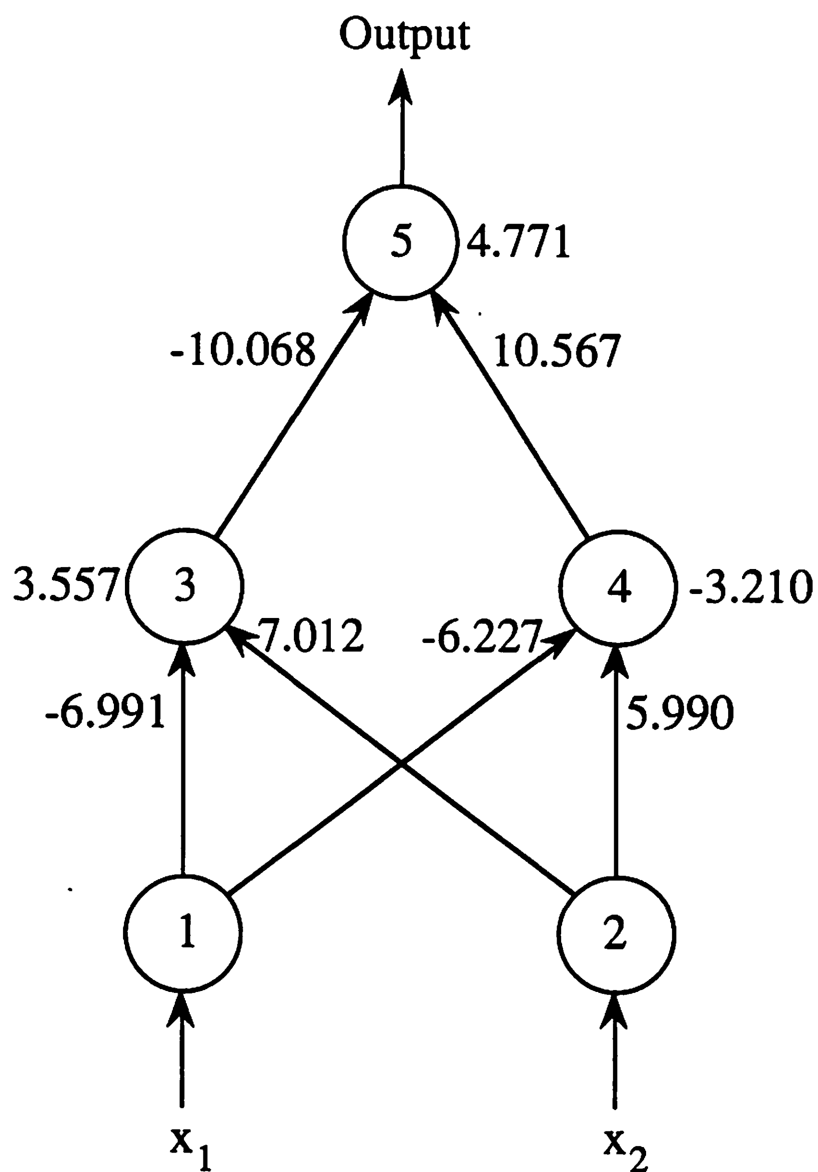


Figure 25. Neural network trained to solve the XOR mapping problem, starting with two hidden neurodes

in the input vector. This network has learned to separate each of the XOR conditions using the two hidden neurodes as detectors for the exclusiveness of the input vector and using the output neurode as the OR.

7.6.1.2 Using Ash's DNC for the XOR problem The second method in training a neural network to perform the XOR mapping employs Dynamic Node Creation as suggested by Ash [11]. In this method, learning begins with a single hidden neurode. Additional neurodes are added as the search stagnates at unacceptably high MSE values. After the addition of a new neurode, learning continues on all of the weights in the network. In performing this training, I used a value of 0.05 for the trigger slope, Δ_T, and a window of 2000 iterations. The network after the first stage of training, with a single hidden neurode, is shown in Fig. 26a and the output values for each training pair are given in Table 18.

Table 18. Output values for each training input-output pair and every non-input neurode in Fig. 26a

Input x_1	Input x_2	Output for neurode #3	Output for neurode #5
0	0	1.000	0.334
0	1	1.000	0.333
1	0	0.205	0.990
1	1	1.000	0.333

It is interesting to see that the single hidden neurode has developed a feature detection mechanism which corresponds roughly to the first hidden neurode (neurode number three) in

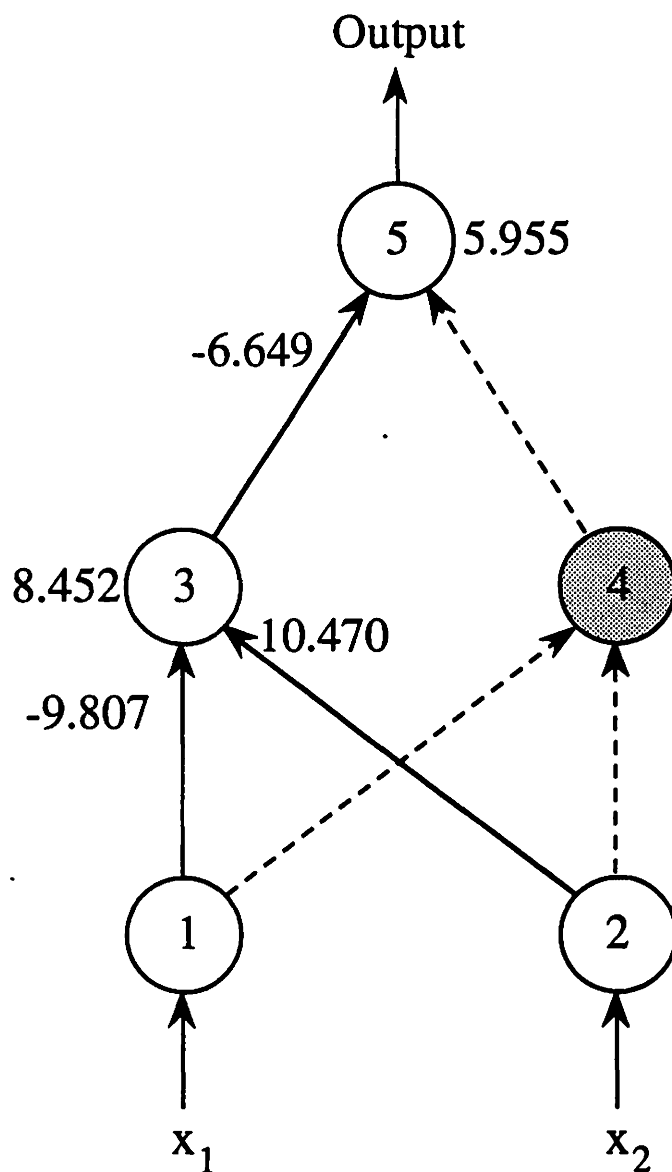


Figure 26a. Intermediate neural network in the process of being trained to solve the XOR mapping problem, starting with one hidden neurode and using Ash's method of DNC.

the network which was trained starting with two hidden neurodes. The one difference is that when the (1,0) feature is detected in the input, the neurode's output is not "turned off" as completely as in the previous network. This is because the network is attempting the minimize the MSE over all patterns, not just this one pattern. As such, the network is still attempting to adjust the weights in a manner which will produce correct results for all of the input-output patterns, even though this is not possible. Therefore, some of the feature detection capabilities of the network are sacrificed in trying to minimize error. In other words, the network is attempting to use too few neurodes in developing feature detectors which will allow the network to correctly classify all input vectors.

Once the trigger slope is crossed, and the width of the window has been surpassed, another hidden neurode is added with random weights. After more training on all of the weights, the network in Fig. 26b is obtained with the corresponding output values shown in Table 19.

Table 19. Output values for each training input-output pair and every non-input neurode in Fig. 26b

Input x_1	Input x_2	Output for neurode #3	Output for neurode #4	Output for neurode #5
0	0	0.999	0.051	0.010
0	1	1.000	0.939	0.990
1	0	0.024	0.000	0.990
1	1	0.999	0.040	0.009

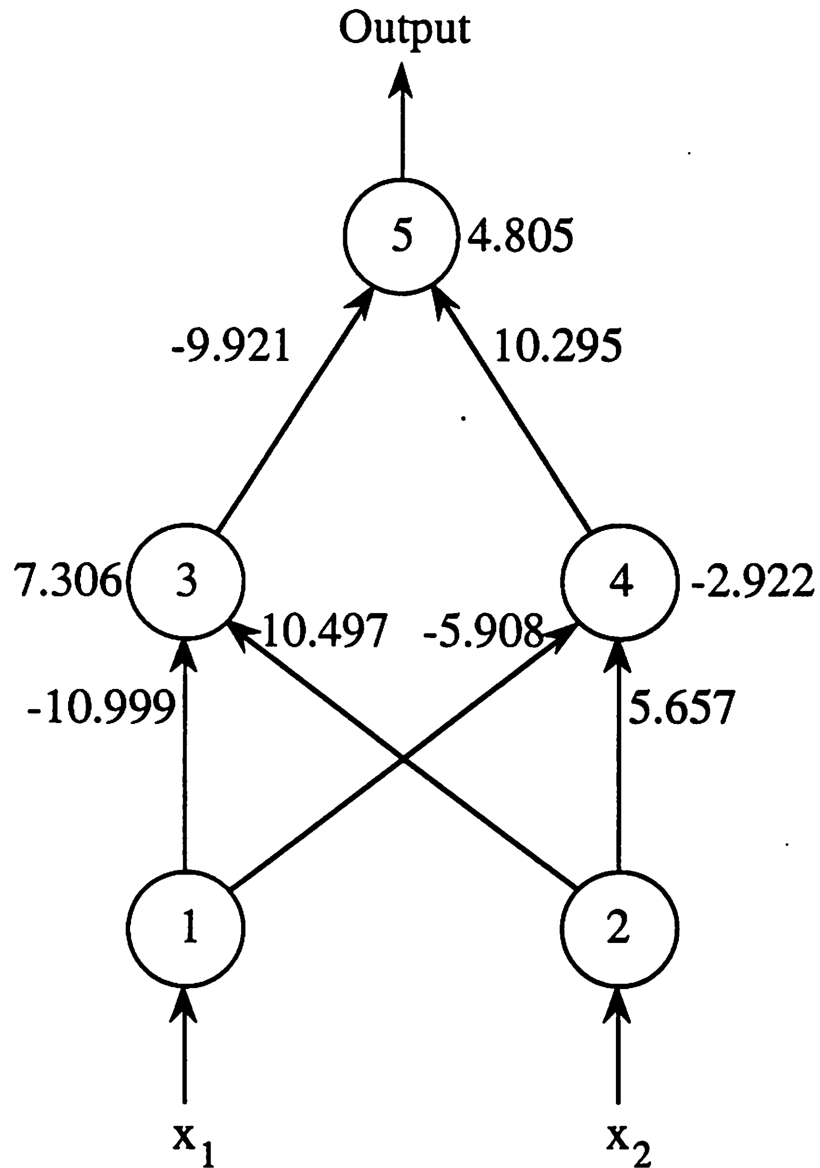


Figure 26b. Neural network trained to solve the XOR mapping problem, starting with one hidden neurode and using Ash's method of DNC during training.

Again, neurode number four (the new neurode) has learned to detect the input pattern (0,1) while neurode number three (the old neurode) has retained the detection of the input pattern (1,0). Notice that the weights on neurode three have changed slightly after the addition of the new neurode. This is the fine tuning of the weights alluded to earlier. Now, neurode number three is "turned off" to greater extent when the pattern (1,0) is present at the inputs. However, the feature detection characteristics of this neurode remain the same--only the degree to which the neurode is turned off when the feature occurs has changed.

7.6.1.3 Using my DNC for the XOR problem The final method of developing a network for solving the XOR problem is based on my method of fixing the old hidden neurode weights and allowing training to continue only on the newly added neurode and on the output neurodes' weights. In doing so, the same initial values for the weights as used in the Ash example were used here, as were the value of the trigger slope and the window. After the first stage of training, the network with a single hidden neurode was, of course, the same as in Section 7.6.1.2 (see Fig. 26a and Table 18). After adding the new neurode, training was performed only on the weights connected to neurodes number four and five. The resulting network is shown in Fig. 27 and the output values are given in Table 20.

Even though the feature detection capabilities of neurode three are not quite as sound as desirable (as in the previous examples), the network has been able to adjust through overcompensation of the weight connecting neurode three to the output neurode. This is why it is important that all of the weights connected to the output neurodes continue to be trained. It allows the output neurodes to adjust how the feature detectors trained earlier in the learning period are combined with newer feature detectors to produce better output responses.

Table 20. Output values for each training input-output pair and every non-input neurode in Fig. 27

Input x_1	Input x_2	Output for neurode #3	Output for neurode #4	Output for neurode #5
0	0	1.000	0.036	0.010
0	1	1.000	0.932	0.992
1	0	0.205	0.000	0.988
1	1	1.000	0.027	0.009

7.7 Conclusions and Discussion

I have used both methods of DNC in training networks to perform the test mappings used in Chapter 6 (parity 3, symmetry 4, encoder 4, adder 2) with great success. In nearly all cases, optimal network architectures have been obtained (for those networks with known optimal sizes). In those that did not result in optimal architectures, the size of the network typically remained within one or two neurodes of the optimal.

Those networks that allowed training to be performed on all weights after the addition of new neurodes quite often produced better results in fewer iterations--at the expense of using more computation in the process. On the other hand, using the method of freezing previously trained neurode weights results in the development of feature detectors which are similar to those found with the previous method. Also, if all of the weights are allowed to train briefly just before the addition of a neurode is to take place, this can often give better results by providing some amount of fine tuning as the learning process continues. In this way, fewer calculations are still required than in the first method, but the weights can be fine tuned slightly before proceeding. I have found that this method works quite well and most often works better

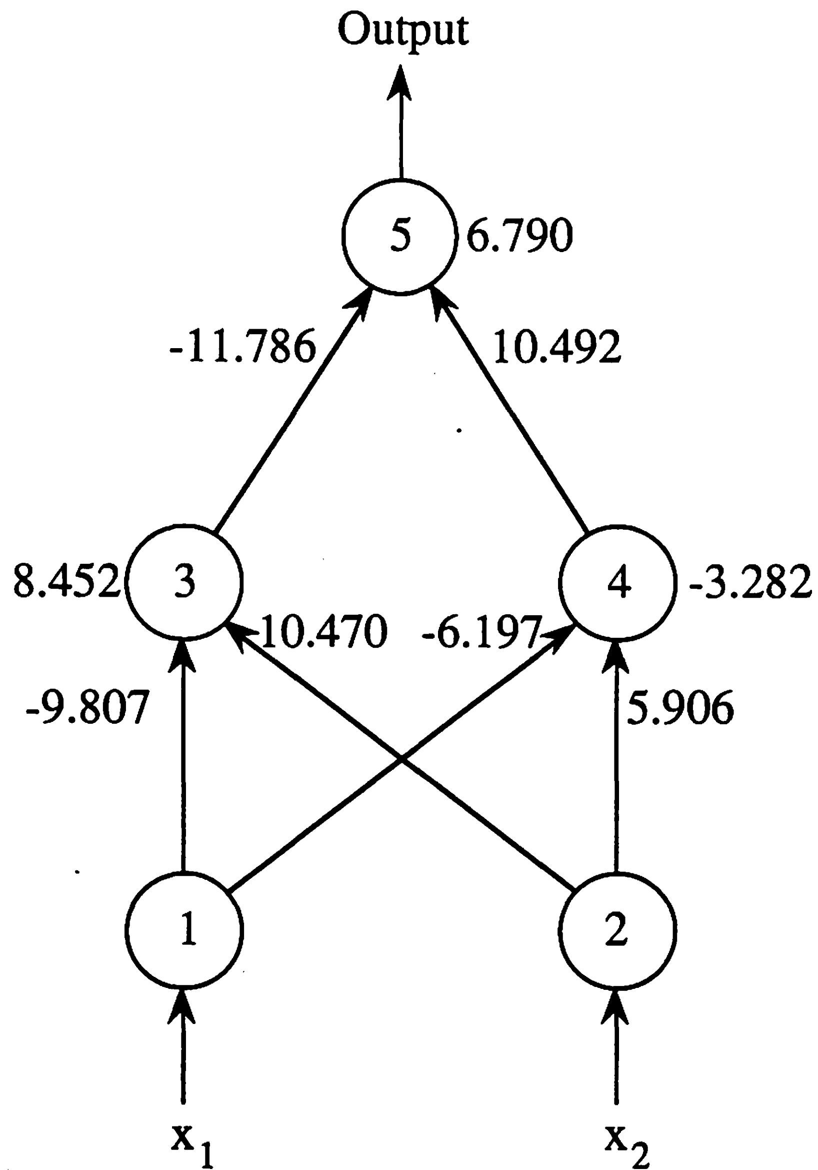


Figure 27. Neural network trained to solve the XOR mapping problem, starting with one hidden neurode and using my alternate method of DNC during training.

than freezing old weights permanently. At the least, when a suitable architecture has been discovered, all of the weights should be allowed to train briefly before declaring the simulation complete.

Using this method of training networks can greatly reduce the "real" time involved in training networks. As alluded to in Chapter 6 when discussing the difficulties in using GBL to solve the adder 2 problem, this method may produce better results in a shorter period of time.

In conclusion, the method of dynamically altering the structure of a network during training can provide good solutions. Until such time as a formal theory on the correct network architecture for any particular mapping problem is provided, methods such as DNC seem to be a viable option for discovering good networks. Further, such methods can often require less computation and less time in training networks, making them even more attractive.

8 SUMMARY AND DISCUSSION

8.1 Overview

This chapter provides a summary and discussion of the three major topics of this thesis, genetic-based learning in artificial neural networks, network architecture, and the application of neural networks to a problem in nondestructive evaluation (NDE)--inversion of uniform field eddy current to obtain flaw sizes. Section 8.2 is a summary of the research results presented in this thesis and Section 8.3 is a discussion of the results and possible future work in these areas.

8.2 Summary

This section provides a summary of the research work presented in Chapters 4-7.

8.2.1 Inversion of uniform field eddy current data

This research work originally began with a desire to show the feasibility of creating a robust, effective, and easy-to-use method of interpreting flaw signals obtained from eddy current probes. In particular, the desire was to obtain quantitative information, actual flaw

dimensions, from these signals. It was felt that previous model-based inversion procedures would not be readily accepted in an industrial setting due to the complexity and understanding of electromagnetic theory required of these approaches. Therefore, an artificial intelligence approach was pursued. Expert systems were eliminated due to a lack of good rules for interpreting the signals. Because neural networks were just coming into the spotlight again and possessed several characteristics which seemed desirable for this work, this is the avenue that was explored. I feel that neural networks have shown great promise in helping to solve the inversion problem for eddy current flaw signals. Certainly the results presented in this thesis (Chapter 4) are very encouraging, even though they represent a limited set and range of flaw sizes.

8.2.2 Genetic-based learning

As the work on eddy current flaw signal interpretation progressed, I quickly found several limitations and problems in using the backpropagation learning algorithm (Chapter 3). It was suggested to me by Dr. Les Schmerr that genetic algorithms, as a global optimization technique, might be able to produce better networks by avoiding local minima. As my investigations into how to use this technique for optimizing neural networks progressed, I recognized several other advantages which GAs might provide over backpropagation and to date these advantages have held. Again, these advantages include a global optimization procedure, arbitrary network architecture, arbitrary activation functions, and a natural parallelization of the learning algorithm.

8.2.3 Network architecture

Finally, one serious problem I found in using neural networks, not related to any particular learning algorithm, is that of determining the proper sized network--number of hidden neurodes to use--for any given problem. Several methods of obtaining networks which were closer to the "optimal" size were investigated in the later stages of the eddy current inversion work. These methods included network pruning [35] and Dynamic Node Creation (DNC) [11]. After much work with both of these, I felt that the DNC method held more promise for obtaining minimally sized networks, while at the same time requiring less computation. I felt this method was especially promising in training networks with genetic-based learning because large networks require very long genotypes, thus incorporating more effort in evaluating each population. If DNC was a feasible alternative (to that of starting with a certain number of hidden neurodes and hoping that the network was of "good" dimension), then this method should alleviate some of the long processing times for larger final networks. However, using the method of DNC discussed by Ash, the reduction in processing time would occur only near the beginning of the training cycle when the number of hidden neurodes was low. Therefore, in order to keep the amount of processing low throughout the training cycle, I began looking into the feature detection mechanisms that may be inherent with a method such as DNC. In analyzing the solutions found by three different methods of training--standard learning with a given number of hidden neurodes, Ash's method of DNC (allowing all weights to train at each stage), and my method of DNC (allowing only new neurode weights to train at each stage)--I discovered the solutions to be remarkably similar. In fact, it was apparent to me that using my method of DNC produced feature detectors in the hidden layer which were nearly as good as those found by the other two methods. When the feature detectors developed with

my method were not quite as good, using one or two more hidden neurodes nearly always provided the additional features required.

8.3 Discussion and Suggestions for Future Work

This section provides a discussion of the research results and gives some suggestions for areas of future work for each of the topics.

8.3.1 Inversion of uniform field eddy current data

Although results presented in Chapter 4 are promising and encouraging, there remains one limitation of the approach which has yet to be overcome. This limitation is developing a neural network which is robust enough to be used on all types of eddy current probes and all types of conducting materials. Currently, in order for eddy current measurements to be meaningful, a calibration must be made with the probe to be used and on the type of material under investigation. Accurate measurements of the calibration factor are certainly possible, but are usually difficult. Further, the types of probes most widely used in an industrial setting are not uniform field probes. This raises several questions about how a network, or a system, might be developed which is robust enough to be used universally, regardless of the probe and material.

I have several suggestions for how this might be accomplished. The first is to develop a separate network for each type of probe on each material type. In practice, simply selecting the probe and material type on the front panel of an instrument would provide for using the correct

network. All that is required is to make a calibration scan in order to properly preprocess the input data or scale the network. Obviously, this requires developing many separate networks and therefore involves many training measurements to be made. While not very elegant, this is the most straightforward approach. Another idea, which simply reduces the number of measurements to be made is to use the model-based inversion procedures to develop training and testing sets. Finally, the most elegant approach would require much theoretical development. This involves developing a mapping between each of the probe types and material types which could alter the network, or change the preprocessing of the input data to correctly compensate for the probe and material. This necessitates training only one network, but would require much effort in determining how one probe operating on one material maps onto another probe on another material.

8.3.2 Genetic-based learning

Although the simulations performed by me have shown a great deal of promise, and often work faster and reduce the MSE to much lower levels than backpropagation, I see three major areas of genetic-based learning which need further exploration. The first is determining exactly the relationship between the parameters used in GBL--how each changes the performance of the algorithm and how the combination affects the outcome of the learning cycle. These parameters include the crossover rate, the mutation rate and/or upper and lower bounds of the mutation rate, the population size, the analog weight range, the number of bits used to encode a weight in the network, the weight resolution factor (determined by the weight range and the number of bits used to encode a weight), and the number of crossover points. Although studies have been made investigating how some of these parameters combine and what the

effects are in situations not related to optimizing neural networks [40], a definitive answer did not result--only guidelines and rules-of-thumb.

The second area of genetic-based learning I would like to see more work done in is that of the parallel algorithm. To date, I have only been able to run this algorithm on an Apollo DN10040 workstation with four processors. Again, I have seen linear performance increase with this parallel algorithm over a serial algorithm. However, I know that as the number of processors increases, this linear performance curve will fall off. I feel this will happen as the overhead in communications between the processors approaches that expended by each of the processors in processing its subpopulation. Also, as the size of the population grows, the serial work done in selection, reproduction, and creating the mating pool will grow, thus degrading the performance somewhat. Therefore, I would like to see this algorithm implemented on a massively parallel machine and have its performance evaluated.

Since the development of my parallel algorithm, I have seen other implementations [50-53] which operate more on each of the subpopulations, even in performing selection, reproduction, and mating. However, these implementations rely on exchanging individuals between the subpopulations, inducing more communications overhead. Further, these implementations do not use the entire population as effectively as my implementation because the exchange of individuals occurs only between certain processors, not all processors, during each generation.

The third area of genetic-based learning which I believe warrants investigation is using this algorithm to optimize more than just the weights in a network, as mentioned briefly at the end of Chapter 5. Possible ares include dynamically configuring the connections, the number of neurodes, the activation functions employed by each of the neurodes, and the parameters used by each of the activation functions.

8.3.3 Network architecture

While I sense that dynamically configuring neural network architectures provides a good means of determining a network's dimension for arbitrary mapping problems, I would like to see more work done in investigating the role of this method in developing good feature detectors. I believe this to be an important area of research for three reasons. The first is in extracting information from a network pertaining to its reasoning in making a conclusion. With a minimal set of good feature detectors, the best rules possible can be developed. Second, due to the lack of a theory for determining the optimal network dimension for any given problem, this method may provide the best heuristic method for discovering good network architectures. Third, if this method works as well over a large variety of problems as it does over the cases I have tested, then reductions in learning times for large problems may be significant.

9 BIBLIOGRAPHY

[1] Mollenhoff, C. R. Atanasoff: forgotten father of the computer. 1st ed. Ames, IA: Iowa State University Press, 1988.

[2] Minsky, M. L., and Papert, S. A. Perceptrons. Prologue. Expanded Edition. Cambridge, MA: M.I.T. Press, 1988.

[3] Nugen, S. M. An expert system for ultrasonic flaw classification. Master's Thesis, Iowa State University, 1989.

[4] Christensen, K. E. An intelligent feature extractor for ultrasonic signals. Master's Thesis, Iowa State University, 1988.

[5] Hecht-Nielsen, R. Neurocomputing. Reading, MA: Addison-Wesley Publishing Co., Inc., forthcoming.

[6] Wasserman, P. D. Neural Computing: Theory and Practice. New York, New York: Van Nostrand Reinhold, 1989.

[7] Pao, Yoh-Han. Adaptive Pattern Recognition and Neural Networks. Reading, MA: Addison-Wesley Publishing Co., Inc., 1989.

[8] Sejnowski, T. J., and Rosenberg, C. R. "Parallel networks that learn to pronounce English text." Complex Systems 3: 145-168, 1987.

[9] Ahalt, S. C., Garber, F. D., Jouny, I., Krishnamurthy, A. K. "Performance of Synthetic Neural Network Classification of Noisy Radar Signals." Advances in Neural Information Processing Systems, ed. D. S. Touretzky. Vol. 1, pp. 281-288. San Mateo, CA: Morgan Kaufmann Publishers, 1989.

[10] Mann, J. M., Schmerr, L. W., Moulder, J. C., Kubovich, M. W. "Inversion of Uniform Field Eddy Current Data Using Neural Networks." Review of Progress in Quantitative NDE, eds. D. O. Thompson and D. E. Chementi. Vol. 9. New York, New York: Plenum Press, forthcoming.

[11] Ash, T. "Dynamic Node Creation in Backpropagation Networks." ICS Report 8901. Institute for Cognitive Science, University of California, San Diego, Feb. 1989.

[12] McCulloch, W. S., and Pitts, W. "A logical calculus of the ideas immanent in nervous activity." Bulletin of Mathematical Biophysics 5: 115-133, 1943.

[13] McCulloch, W. W., and Pitts, W. "How we know universals." Bulletin of Mathematical Biophysics 9: 127-147, 1947.

[14] Rosenblatt, F. "Two theorems of statistical separability in the perceptron." Proceedings of a Symposium on the Mechanization of Thought Processes 421-456. London: Her Majesty's Stationary Office, 1959.

[15] Rosenblatt, F. Principles of Neurodynamics. New York: Spartan Books, 1962.

[16] Rumelhart, D. E., Hinton, G. E. and Williams, R. J. "Learning Internal Representations by Error Propagation." Parallel Distributed Processing: Exploration in the Microstructure of Cognition. Vol. 1. Chap. 8. Cambridge, MA: M.I.T. Press, 1986.

[17] Werbos, P. J. Beyond Regression: New tools for prediction and analysis in the behavioral sciences. Master's Thesis, Harvard University, 1974.

[18] Parker, D. B. "Learning Logic." Invention Report S81-64, File 1. Office of Technology Licensing. Stanford University, 1982.

[19] LeCun, Y. "Une procedure d'apprentissage pour reseau a seuil assymetrique [A learning procedure for asymmetric threshold network]." Proceedings of Cognitiva 85: 599-604, June 1985.

[20] Stornetta, W. S., and Huberman, B. A. "An improved three-layer, backpropagation algorithm." Proceedings of the IEEE First International Conference on Neural Networks, eds. M. Caudill and C. Butler. San Diego, CA: SOS Printing, 1987.

[21] Wasserman, P.D. "Combined backpropagation/Cauchy machine." Proceedings of the International Neural Network Society. New York: Pergamon Press, 1988.

[22] Sutton, R. S. "Two problems with backpropagation and other steepest-descent learning procedures for networks." Proceedings of the Eighth Annual Conference of the Cognitive Science Society, pp. 823-831, 1986.

[23] Jacobs, R. A. "Increased Rates of Convergence Through Learning Rate Adaptation." Neural Networks 1, No. 4: 295-307, 1988.

[24] Hush, D. R., and Salas, J. M. "Improving the Learning Rate of Backpropagation with the Gradient Reuse Algorithm." IEEE International Conference on Neural Networks 1: 441-448, July 1988.

[25] Owens, A. J., and Filkin, D. L. "Effecient Training of the Back Propagation Network by Solving a System of Stiff Ordinary Differential Equations." IJCN International Joint Conference on Neural Networks 2: 381-386, June 1989.

[26] Halmshaw, R. Non-destructive testing. London: Edward Arnold, 1987.

[27] Auld, B. A., S. R. Jefferies, and J. C. Moulder. "Eddy-Current Signal Analysis and Inversion for Semielliptical Surface Cracks." Journal of Nondestructive Evaluation 7: 79-94, June 1988.

[28] Sabbagh, L. D. and H. A. Sabbagh. "Eddy-Current Modeling and Flaw Reconstruction." Journal of Nondestructive Evaluation 7: 95-110, June 1988.

[29] Udpa, L. and S. S. Udpa. "Solution of Inverse Problems in Eddy-Current Nondestructive Evaluation (NDE)." Journal of Nondestructive Evaluation 7: 111-120, June 1988.

[30] Auld, B. A., F. G. Muennemann, and M. Riaziat. "Quantitative Modelling of Flaw Responses in Eddy Current Testing." Research Techniques in Nondestructive Evaluation. Vol. VII. Chap. 2. London: Academic Press, 1984.

[31] Auld, B. A., S. R. Jefferies, J. C. Moulder, and J. C. Gerlitz, "Semi-elliptical Surface Flaw EC Interaction and Inversion: Theory." Review of Progress in Quantitative NDE, eds. D. O. Thompson and D. E. Chementi. Vol. 5. pp. 383-393. New York, New York: Plenum Press, 1985.

[32] Smith, E. "Characterization of EDM Notches and Real Fatigue Cracks in Flat Surfaces Using Uniform Field Eddy Current Technique." Materials Evaluation 43: 1640-1643, Dec. 1985.

[33] Moulder, J. C., P. J. Shull, and T. E. Capobianco. "Uniform Field Eddy Current Probe: Experiments and Inversion for Realistic Flaws." Review of Progress in Quantitative NDE, eds. D. O. Thompson and D. E. Chementi. Vol. 6. pp. 601-610. New York, New York: Plenum Press, 1986.

[34] Shull, P. J., T. E. Capobianco, and J. C. Moulder. "Design and Characterization of Uniform Field Eddy Current Probes." Review of Progress in Quantitative NDE, eds. D. O. Thompson and D. E. Chementi. Vol. 6. pp. 695-703. New York, New York: Plenum Press, 1986.

[35] Sietsma, J. and Dow, R. J. F. "Neural Net Pruning -- Why and How." IEEE International Conference on Neural Networks 1: 325-333, July 1988.

[36] Goldberg, D. E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley Publishing Co., Inc., 1989.

[37] Whitley, D., and Hanson, T. "The Genitor Algorithm: Using Genetic Recombination to Optimize Neural Networks." Technical Report CS-89-107. Department of Computer Science, Colorado State University, Boulder, 1989.

[38] Holland, J. H. "Schemata and intrinsically parallel adaptation." Proceedings of the NSF Workshop on Learning System Theory and its Applications, eds. K. S. Fu and J. S. Tou. University of Florida, Gainesville, 1973.

[39] Holland, J. H. Adaptation in Natural and Artificial Systems. University of Michigan, Ann Arbor, 1975.

[40] Grefenstette, J. J. "Optimization of Control Parameters for Genetic Algorithms." IEEE Transactions on Systems, Man, and Cybernetics SMC-16, No. 1: 122-128, Jan./Feb. 1986.

[41] Baker, J. E. "Adaptive Selection Methods for Genetic Algorithms." Proceedings of the First International Conference on Genetic Algorithms and their Applications, ed. John J. Grefenstette. pp. 101-111. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1988.

[42] Goldberg, D. E., and Richardson, J. "Genetic Algorithms with Sharing for Multimodal Function Optimization." Proceedings of the First International Conference on Genetic Algorithms and their Applications, ed. John J. Grefenstette. pp. 41-49. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1988.

[43] Fukushima, K. "Neocognitron: A Self-organizaing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position." Biological Cybernetics 36: 193-202, 1980.

[44] Gutierrez, M., Wang, J., and Grondin, R. "Estimating Hidden Unit Number for Two-layer Perceptrons." IJCN International Joint Conference on Neural Networks 1: 677-681, June 1989.

[45] Hecht-Nielsen, R. "Kolmogorov's Mapping Neural Network Existence Theorem." Proc. 1987 IEEE International Conference on Neural Networks III: 11-13. New York: IEEE Press, 1987.

[46] Kolmogorov, A. N. "On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition." Dokl. Akad. Nauk USSR 114: 953-956. 1957.

[47] Lorentz, G. G. "The 13-th Problem of Hilbert." Proc. of Symposia in Pure Math. 28. American Mathematical Society, Providence, R.I., 1976.

[48] Gallant, S. I. "Connectionist Expert Systems." Communications of the ACM 31, No. 2: 152-169, Feb. 1988.

[49] Lippman, R. P. "An Introduction to Computing with Neural Nets." IEEE ASSP Magazine 4, No. 2: 4-22, April 1987.

[50] Pettey, C. B., Leuze, M. R., and Grefenstette, J. J. "A parallel genetic algorithm." Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms, ed. John J. Grefenstette. pp. 155-161. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1987.

[51] Sannier, A. V., II, and Goodman, E. D. "Genetic learning procedures in distributed environments." Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms, ed. John J. Grefenstette. pp. 162-169. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1987.

[52] Jog, P., and Van Gucht, D. "Parallelisation of probabilistic sequential search algorithms." Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms, ed. John J. Grefenstette. pp. 170-176. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1987.

[53] Tanese, R. "Parallel genetic algorithms for a hypercube." Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms, ed. John J. Grefenstette. pp. 177-183. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1987.

10 ACKNOWLEDGEMENTS

First, and foremost, I would like to thank those persons who have helped me and given me advice and encouragement in performing this research. Those persons include the members of my committee, Dr. Charles T. Wright, Jr., Dr. Lester W. Schmerr, Jr., and Dr. Terry A. Smay. I very much appreciate the freedom and responsibility given to me, especially by Dr. Schmerr, in allowing me to direct the research in a manner I felt necessary. I would also like to thank John Moulder for his help and patience in tutoring me on the finer points of eddy current measurements, as well as for his time in reviewing several of my manuscripts. I am also indebted to my friend and roommate, Mark Kubovich, for his help in taking measurements and for the many discussions we have had about nearly everything. Finally, I would like to thank Dr. Donald O. Thompson for ultimately giving me the opportunity to pursue this research at the Center for NDE.

Of course, I am forever grateful to my parents for stressing the importance of a good education and strong work ethic, and for their help (both monetarily and psychologically) in allowing me to pursue this.

I would like to extend special thanks to Dr. Michael Hughes, without whom I would have never had the opportunity to be involved with this work. His initial hiring of me two summers ago on a separate project helped me immensely in obtaining support for this research.

Having had the good fortune of being able to work with Mike has allowed me to grow a great deal as a person.

I would also like to thank Steve Nugen, Chien-Ping Chiou, Brian Lovewell, and Sung-Jin Song for their friendship and the many discussions I have had with each of them about neural networks, artificial intelligence, computers, etc.

And last, but certainly not least, I would like to thank my fiancee, Pam, for her undying love and friendship during the course of this research--late nights, weekends, holidays, and all. In no way will I ever be able to fully express my appreciation for her understanding and commitment.

This research was supported by the Center for Nondestructive Evaluation at Iowa State University and was performed at Ames Laboratory which is operated for the U.S. Department of Energy by Iowa State University under Contract No. W-7405-ENG-82.