Spring 1992

Characterizing and improving the fault tolerance of artificial neural networks

Bruce Edmond Segee
University of New Hampshire, Durham

Order Number 9225264

Characterizing and improving the fault tolerance of artificial neural networks

Segee, Bruce Edmond, Ph.D.

University of New Hampshire, 1992

Characterizing and Improving the Fault Tolerance of Artificial Neural Networks

BY

Bruce E. Segee

BSEE University of Maine, 1985
MSEE University of Maine, 1989

Dissertation

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

in

Engineering

May, 1992

This dissertation has been examined and approved.

Michael J Carter

Dissertation director, Michael J. Carter, Assistant Professor of Electrical Engineering

[illegible]

W. Thomas Miller III, Professor of Electrical Engineering

Filson H. Glanz

Filson H. Glanz, Professor of Electrical Engineering

[illegible]

Lee L. Zia, Associate Professor of Mathematics

[illegible]

Barry K. Fussell, Assistant Professor of Mechanical Engineering

April 21, 1992

Date

TABLE OF CONTENTS

List of Figures viii

Abstract xiii

Introduction 1

Chapter I

Artificial Neural Networks 4

1.1 Types of Neural Networks 4

1.1.1 Feedforward Neural Networks 6

1.1.1.1 Multilayer Perceptron 6

1.1.1.2 Generalized Radial Basis Function Network 7

1.1.1.3 CMAC 8

1.1.2 Recurrent Neural Networks 9

1.1.2.1 Simple Recurrent Networks 10

1.1.2.2 Settling Networks 10

1.1.2.3 Neural Oscillators 11

1.2 Uses of Neural Networks 12

1.2.1 Pattern Recognition/Discontinuous Function Approximation 12

1.2.2 Continuous Function Approximation 14

1.3 Fault Tolerance 14

1.3.1 Redundancy 15

1.3.2 Graceful Degradation, Fail Safe/Fail Soft....15
1.3.3 Distribution of Calculation....16

Chapter II

Fault Tolerance of Artificial Neural Networks....18

2.1 Prior work....18

2.2 Formulating a Fault Tolerance Measure....19

2.2.1 Choosing an Error Measure....20

2.2.1.1 Maximum Deviation....20

2.2.1.2 RMS Error....21

2.2.1.3 Approximation Quality....22

2.2.2 Choosing a Faulting Method....22

2.2.2.1 Average or Typical Single Fault....23

2.2.2.2 Most Damaging Single Fault....23

2.2.2.3 Multiple Faults....24

2.2.2.3.1 Combinatorial Growth of Possibilities....24

2.2.2.3.2 Sampling....25

2.2.2.3.3 A Worst Case Path Approach....26

Chapter III

Fault Tolerance of the Multilayer Perceptron....28

3.1 Experiment Definition....28

3.2 Training Methods....29

3.2.1 Backpropagation....29

3.2.2 Backpropagation with Momentum....30

3.2.3 Backpropagation with Flexible Learning Rate 30

3.3 Experiment Results 31

3.3.1 Learning Performance 31

3.3.2 Single Weight Fault Tolerance 33

3.3.3 Worst Case Path Performance 36

3.3.4 Partial Fault Tolerance 38

3.4 The Effect of Network Pruning 41

Chapter IV

Fault Tolerance of a Radial Basis Function Network 45

4.1 Experiment Definition 45

4.2 Training Methods 46

4.2.1 Weights 46

4.2.2 Centers 48

4.2.3 Widths 51

4.3 Experiment Results 52

4.3.1 Learning Performance 53

4.3.2 Single Weight Fault Tolerance 55

4.3.3 Worst Case Path Performance 56

4.3.4 Partial Fault Tolerance 58

Chapter V

Using Spectral Techniques for Analyzing Neural Networks 61

5.1 Function Approximation With Neural Networks 61

5.1.1 Utilizing Linear Superposition 62

5.1.2 Frequency Domain Analysis 63

5.2 Spectra of Common Basis Functions 65

5.2.1 The Sigmoid Function 65
5.2.2 Gaussian Curves 71
5.2.3 The Rectangular Pulse Function 72
5.2.4 The Triangular Pulse Function 73
5.3 CMAC Slow Learning 75
5.4 Experiment Results 77
5.5 Spectral Fitting and Fault Tolerance 81

Chapter VI

Methods for Improving Fault Tolerance 83
6.1 Spectral Methods 83
6.2 Engineered Redundancy 83
6.2.1 Weight Redundancy 84
6.2.2 Node Redundancy 84
6.2.3 Network Redundancy 85
6.2.4 Criticism of Engineered Redundancy 86
6.3 Good Initial Conditions 87
6.4 Training With Intermittent Faults 90
6.4.1 Multilayer Perceptron Results 90
6.4.2 Gaussian Radial Basis Function Results 93
6.4.2.1 Single Weight Fault Tolerance 96
6.4.2.2 Partial Fault Tolerance 100
6.4.3 Comparing the Multilayer Perceptron and the Generalized Radial Basis Function Network 102
6.5 Limiting the Possible Effect of a Fault 105
6.5.1 Limited Dynamic Range Weights 106

6.5.2 Continuous Training....106

6.5.3 Sparse Connectivity....107

Chapter VII

Choosing Network Parameters for Improved Fault Tolerance....109

7.1 Choosing a Suitable Activation Function....109

7.2 Choosing Initial Centers/Biases....112

7.3 Choosing Initial Weights....112

Chapter VIII

Wrap up....114

8.1 Summary and Conclusions....114

8.2 Possible Future Work....119

References....122

LIST OF FIGURES

1.1 A weighted sum processing element 5

1.2 A distance measuring or receptive field processing element 5

1.3 A diagram of a multilayer perceptron 7

1.4 A diagram of a generalized radial basis function network 8

1.5 A diagram of a Hopfield network 11

2.1 Number of combinations of multiple weight faults (up to seven) for a network having fifty weights 25

3.1 RMS error versus training epoch for the multilayer perceptron trained using standard backpropagation. ... 32

3.2 RMS error versus training epoch for the multilayer perceptron trained using momentum 32

3.3 RMS error versus training epoch for the multilayer perceptron trained using a flexible learning rate. ... 33

3.4 RMS error change due to single weight loss in a multilayer perceptron trained with backpropagation ... 34

3.5 RMS error change due to single weight loss in a multilayer perceptron trained with momentum 35

3.6 RMS error change in a multilayer perceptron trained with flexible learning rate 35

3.7 Worst case path AQ curve for the multilayer perceptron trained using standard backpropagation 36

3.8 Worst case path AQ curve for the multilayer perceptron trained using momentum 37

3.9 Worst case path AQ curve for the multilayer perceptron trained using flexible learning rate 37

3.10 Partial fault tolerance of the multilayer perceptron trained using standard backpropagation 39

3.11 Partial fault tolerance of the multilayer perceptron trained using momentum 40

3.12 Partial fault tolerance of the multilayer perceptron trained using flexible learning rate 40

3.13 RMS error change due to single weight loss in an unpruned multilayer perceptron 43

3.14 RMS error change due to single weight loss in a pruned multilayer perceptron 43

4.1 Magnitude of $\frac{\partial F(x)}{\partial c_i}$ as a function of $(x-c_i)$ for a unity weight and width parameter of two 49

4.2 Magnitude of $\frac{\partial F(x)}{\partial c_i}$ as a function of $(x-c_i)$ for a unity weight and width parameter of 0.25 50

4.3 RMS error curve for a Gaussian radial basis function network with 100 units 53

4.4 RMS error curve for a Gaussian radial basis function network with 200 units 54

4.5 RMS error change caused by single weight loss in a 100 unit Gaussian radial basis function network 55

4.6 RMS error change caused by single weight loss in a 200 unit Gaussian radial basis function network 56

4.7 Fault tolerance curve for the 100 unit Gaussian radial basis function network 57

4.8 Worst case path AQ curve for the 200 unit Gaussian radial basis function network 57

4.9 Partial fault tolerance of the 100 unit Gaussian radial basis function network 59

4.10 Partial fault tolerance of the 200 unit Gaussian radial basis function network 60

5.1 A single-input, single-output network 62

5.2 Fourier transform of the sigmoid function with the value 'a' set to 0.25 69

5.3 Fourier transform of the sigmoid function with the value 'a' set to 1.0 70

5.4 Fourier transform of the sigmoid function with the value 'a' set to 4.0 70

5.5 Fourier transform of the Gaussian function for w = 0.25, w = 1, and w = 4. 72

5.6 A plot of the rectangular pulse function 72

5.7 Fourier transform of the rectangular pulse function 73

5.8 A plot of the triangular pulse function 74

5.9 Fourier transform of the triangular pulse function ... 74

5.10 RMS error curve for a 100 unit Gaussian radial basis function network with a width parameter of 4 ... 79

5.11 RMS error change caused by single weight loss in a network having a width parameter of 4.0 79

5.12 Worst case path AQ curve for a network having a width parameter of 4.0 80

6.1 AQ curve for the multilayer perceptron initialized with well chosen weights (10,000 training passes) 89

6.2 AQ curve for the multilayer perceptron initialized with well chosen weights (100,000 training passes) ... 89

6.3 AQ curve for the multilayer perceptron trained with one intermittently failing unit 91

6.4 AQ curve for the multilayer perceptron trained with five intermittently failing units 91

6.5 AQ curve for the multilayer perceptron trained with ten intermittently failing units 92

6.6 AQ curve for the multilayer perceptron trained with twenty intermittently failing units 92

6.7 AQ curve for the GRBF network trained with one intermittently failing unit 94

6.8 AQ curve for the GRBF network trained with five intermittently failing units 94

6.9 AQ curve for the GRBF network trained with ten intermittently failing units 95

6.10 AQ curve for the GRBF network trained with twenty intermittently failing units 95

6.11 Single weight loss RMS error change in the GRBF network trained with one intermittently failing unit 97

6.12 Single weight loss RMS error change in the GRBF network trained with five intermittently failing units 98

6.13 Single weight loss RMS error change in the GRBF network trained with ten intermittently failing units 98

6.14 Single weight loss RMS error change in the GRBF network trained with twenty intermittently failing units 99

6.15 Partial fault tolerance for the GRBF network trained with one failing unit 101

6.16 Partial fault tolerance for the GRBF network trained with twenty failing units 101

6.17 Single weight loss RMS error change in a multilayer perceptron trained with 20 intermittently failing units 104

6.18 Partial fault tolerance in a multilayer perceptron trained with 20 intermittently failing units 105

ABSTRACT

Characterizing and Improving the Fault Tolerance of Artificial Neural Networks

by

Bruce E. Segee
University of New Hampshire, May, 1992

Artificial neural networks are networks of very simple processing elements based on an approximate model of a biological neuron. It is widely believed that because biological neural networks are tolerant of the loss of individual neurons and because there is a strong analogy between biological neural networks and artificial neural networks, then artificial neural networks must also be inherently fault tolerant. This is, unfortunately, simply not true.

Results reported in this dissertation show that in the task of function approximation the multilayer perceptron is very intolerant of faults to the extent that the loss of a single network parameter can ruin the learned approximation. A method for quantitatively evaluating network fault tolerance is proposed in this dissertation. This method is applicable to a large number of artificial neural network architectures. Using this method, it can be shown that the generalized radial basis function network is much more fault tolerant than the multilayer perceptron. Additionally, the generalized radial basis function network learns more rapidly than the multilayer perceptron.

These findings can be explained by using spectral methods. When the spectral content of a network's activation function is not similar to the spectral content of the function to be learned, then learning is made very difficult and the learned solution is very sensitive to the loss or alteration of network parameters.

Numerous methods for improving the fault tolerance of artificial neural networks are presented and discussed. Interestingly, experimental results show that a well chosen set of initial conditions can measurably improve fault tolerance. Also, training with random intermittent faults can significantly improve the fault tolerance of neural networks. In fact, the improvement in fault tolerance in the generalized radial basis function network was such that the loss of any single weight actually caused improvement or no change from the fault-free performance. Finally, this dissertation presents guidelines for intelligently choosing network parameters for good learning as well as improved fault tolerance.

INTRODUCTION

The field of artificial neural networks is one which shows great potential. Indeed, artificial neural networks are already routinely used to solve complex, nonlinear problems. Unfortunately, several properties of neural networks are poorly understood. One such property is fault tolerance.

Artificial neural networks are generally modeled after biological neural networks. The rationale for this is that biological systems are capable of solving vastly complex problems such as pattern recognition, control of systems with many degrees of freedom (such as arms and legs), and speech recognition, not to mention thinking and associative recall. One would naturally hope that artificial neural networks would be able to capture some of the desirable qualities of biological neural networks.

Biological neural networks also show an amazing degree of fault tolerance. For instance, brain cells die throughout one's life without noticeably impairing one's ability to walk or understand speech. In fact, in many cases, even severe trauma to the brain causes only degradation in task performance, rather than a total loss of function.[McCarthy, 1990]

Fault tolerance is desirable in virtually any system. Furthermore, although artificial neural networks are patterned after biological neural networks, they are not inherently fault tolerant [Holmes, 1991, Séquin, 1990, Segee, 1990]. This dissertation reports the results of a quantitative study of the fault tolerance of artificial neural networks and also presents some methods for improving the fault tolerance of neural networks.

Chapter I contains an overview of many of the types of artificial neural networks, as well as some typical uses for artificial neural networks. It also discusses what is meant by fault tolerance.

In Chapter II previous work in the field of fault tolerance of artificial neural networks is reviewed. In addition, various considerations in measuring the fault tolerance of artificial neural networks are discussed, e.g., whether to consider worst case or average performance.

In Chapter III results are presented of a study of the fault tolerance of a multilayer perceptron, arguably the most commonly used artificial neural network. This study concentrated on single-input single-output networks trained for function approximation. The principle finding was that the multilayer perceptron is not inherently fault tolerant; Critical weights form during training which, when lost, destroy network performance.

In Chapter IV results are presented of a study similar to that of Chapter III, but using Gaussian radial basis function networks. The findings were that these networks performed significantly better than the multilayer perceptron, both in terms of fault tolerance and training time.

In Chapter V a technique is discussed which uses frequency domain information in order to predict both the fault tolerance of a network and the difficulty that a network will have in learning a particular problem. Conversely, this method may be used to choose a good neural activation function which gives rapid learning, good performance, and fault tolerance.

In Chapter VI methods are discussed by which the fault tolerance of artificial neural networks may be improved. Frequency domain methods of Chapter V are discussed as are redundancy, intelligent choice of initial conditions, training with faults as a means of improving fault tolerance, and limiting the possible effect of a fault.

In Chapter VII some considerations are presented for achieving good network performance, both in terms of approximation error and fault tolerance, for a wide range of problems. These considerations include choice of network activation function, receptive field placement, and initial weight values.

Finally, Chapter VIII provides a summary of the dissertation and discusses possible future work.

CHAPTER I

ARTIFICIAL NEURAL NETWORKS

This chapter is intended to provide the reader with some background in neural networks and fault tolerance. Several different types of neural networks are discussed. Following this, some uses of neural networks are discussed. Finally, several key topics in fault tolerance are discussed.

1.1 Types of Neural Networks

Artificial neural networks typically are formed from individual processing elements. These processing elements are connected together to form networks. Generally, these processing elements perform one of two functions.

The first type of processing element is illustrated in Figure 1.1. This processing element receives one or more inputs X, each of which is weighted by some value W. These weighted values are summed, a bias value, θ, is subtracted, and the result is operated on by some function, $f(\bullet)$. Typical functions are hard-limiting thresholds and sigmoids.

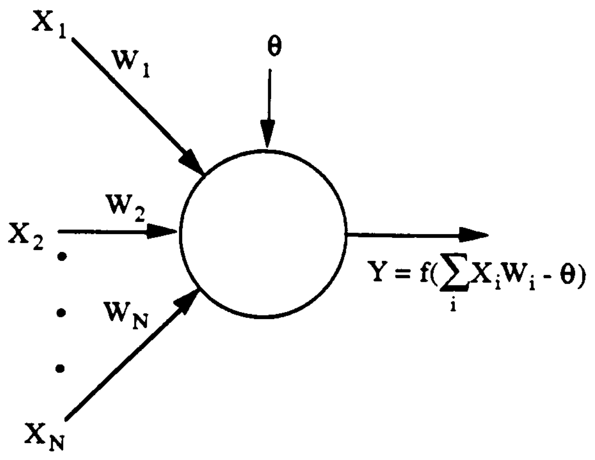

Figure 1.1 A weighted sum processing element

The second type of processing element is shown in Figure 1.2. This processing element, like the first type, has one or more inputs X. However, unlike the first type of processing element, this processing element has an associated location in the input space. The value produced by this processing element is a function of the distance of the current input X from the processing element's location in space C. Typical activation functions are square pulses, triangular pulses, and Gaussian "bumps".

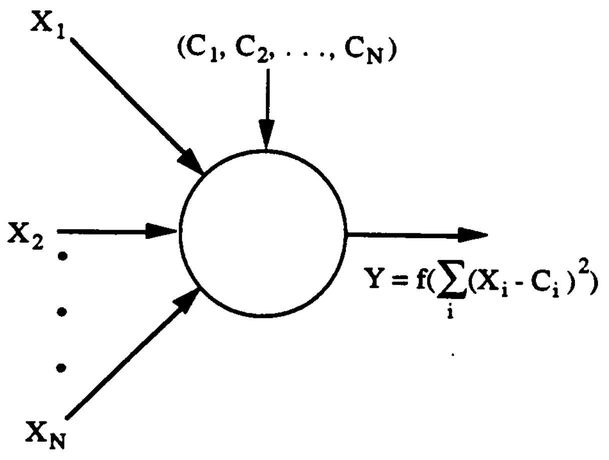

Figure 1.2 A distance measuring or receptive field processing element.

Generally speaking, all neural networks can be divided into two broad categories, feedforward and recurrent. These are discussed separately.

1.1.1 Feedforward Neural Networks

Feedforward neural networks are those for which the current network output is only a function of the current network input. In other words, signals propagate forward from the input to the output. Although there are many different types of feedforward neural networks, three typical feedforward neural networks will be discussed here. They are the multilayer perceptron, the generalized radial basis function network, and the CMAC.

1.1.1.1 Multilayer Perceptron

The multilayer perceptron (see Lippmann's review article [Lippmann, 1988] for a more thorough treatment) is a network composed of layers of weighted sum processing elements. There are no connections between nodes of the same layer. Nodes receive input from the previous layer and deliver their output to the next layer. The activation function used is typically the sigmoid function.

Figure 1.3 shows a diagram of a multilayer perceptron. This particular network has two layers containing four processing elements each. The network inputs and outputs are shown in solid black.

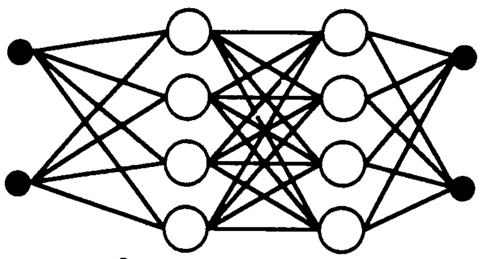
Figure 1.3 A diagram of a multilayer perceptron.

Each connection, represented by a solid line, has a weight associated with it. Additionally, each processing element has a bias (not shown) associated with it. The weights and biases are typically initialized to small random values and adjusted during learning to improve the network's performance.

1.1.1.2 Generalized Radial Basis Function Network

The generalized radial basis function network [Poggio, 1990] is typically composed of a single layer of distance measuring processing elements. The distance measurement is usually euclidean distance, and the activation function is generally Gaussian. When a Gaussian activation function is used these networks are also called Gaussian radial basis function (GRBF) networks.

A diagram of a generalized radial basis function network is shown in Figure 1.4. The network shown has two inputs, two outputs, and four processing elements. The thick lines represent connections which have an adjustable weight associated with them.

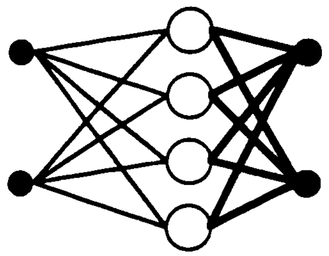

Figure 1.4 A diagram of a generalized radial basis function network.

Each processing element has a center location (which is a vector of the same dimension as the input), and a width parameter which characterizes the spread of the function in the input space. The values of the center and width parameters may be set prior to training or may be adjusted during training.

1.1.1.3 CMAC

CMAC, standing for Cerebellar Model Arithmetic Computer [Albus, 1971, Miller, 1990] is a special type of distance measuring artificial neural network. The CMAC network is very well suited to implementation using digital computer technology [Miller, 1991].

A standard CMAC breaks the input space into hypercubic overlapping receptive fields. Conceptually, each receptive field has an adjustable weight associated with it; However, to keep memory requirements to a reasonable level, available weights are randomly assigned to multiple receptive fields. The receptive fields are arranged in the input space in such a way that each point in the input space activates exactly C receptive fields, where C is called the generalization parameter[1].

CMAC inputs are quantized and thus the network produces a piecewise constant function approximation. Furthermore, CMAC is well suited to fixed point implementations. This, coupled with the fact that only C weights need to be acted upon at any point in time, leads to very fast digital hardware implementations. For a diagram of CMAC see [Miller, 1990].

1.1.2 Recurrent Neural Networks

Recurrent neural networks are those which employ feedback. Thus, in a recurrent neural network, the current network output depends not only on the current input, but also on previous outputs from one or more unit in the network. Recurrent networks may be classified into one of three categories: 1) networks which settle to some stable state; 2) networks which oscillate; and 3) networks which neither settle nor oscillate. For lack of better terms I will refer to these networks as settling networks, neural oscillators, and simple recurrent networks, respectively.

[1]The term "generalization" in this context refers to the overlap of receptive fields as opposed to the ability of the network to correctly respond to novel inputs.

1.1.2.1 Simple Recurrent Networks

A simple recurrent network is a network which has a feedback path from the network output to the input[2]. Thus, the previous output is used in addition to the current input to determine the current output. Virtually any feedforward neural network can be made into a simple recurrent network by simply treating the previous network output as an additional input.

There are a number of reasons why one might choose to use a simple recurrent network. A particularly interesting example is the "truck backer-upper" [Nguyen, 1990] for which an error signal was available only infrequently. The feedback connection allowed the network to be trained even for states which had no explicit error signal by using backpropagation through time [Werbos, 1990].

1.1.2.2 Settling Networks

The best example of a settling network is the Hopfield network (see Lippmann's review article [Lippmann, 1988] for a more thorough treatment). The Hopfield network consists of a single layer of weighted sum processing elements using the sigmoidal activation function. The output of each processing element is connected as an input to every other processing element. Each connection is weighted, and the weights are symmetrical, e.g., the weight from output 2 to input 1 is the same as the weight from output 1 to input 2. Furthermore,

[2]The input, output and feedback path may all be vector quantities.

there are no self-connections. A diagram of a Hopfield network is shown in Figure 1.5.

The network shown has four processing elements (a typical Hopfield network has many more). The output of each node is connected to an input of every other node. Each input has a weight associated with it.

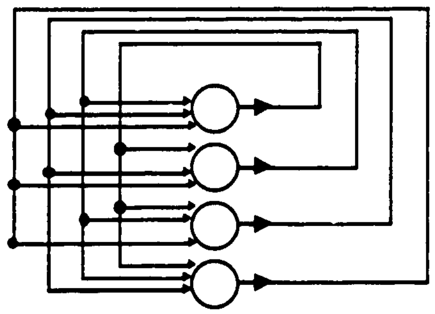

Figure 1.5 A diagram of a Hopfield network

The weights in a Hopfield network are chosen *a priori*, i.e., they are not trained. By appropriately choosing weights, one can induce local minima in an objective function defined on the network's state space. During operation, the network is initialized to some state. The network then settles to the nearest minimum in the state space. This can provide an associative recall function from noisy or partial input data.

1.1.2.3 Neural Oscillators

Unlike a Hopfield network, in which the network settles from the current state to a locally minimum energy state, some recurrent neural networks are designed to oscillate. Oscillations can be useful for learning sequences of actions, such as learning a periodic trajectory for a robot arm

[Jordan, 1988]. There is also evidence that biological systems take advantage of oscillation for coding information [Niebur, 1990, Kruglyak, 1990] and as central pattern generators [Kleinfeld, 1989].

1.2 Uses of Neural Networks

Neural networks have been used in a wide variety of applications ranging from recognition of handwritten digits to playing backgammon. Neural networks essentially perform function approximation, that is, given an input value (and possibly a state value for recurrent networks) they produce an output value which closely approximates a desired response function. The desired response function may be nonlinear and multidimensional. Although pattern recognition can be thought of as a subset of function approximation (i.e., discontinuous function approximation), it is sufficiently common to warrant a separate section.

1.2.1 Pattern Recognition/Discontinuous Function Approximation

Biological systems are very adept at identifying patterns, either visual (such as faces, written text, predators, food, etc.) or audible (such as spoken words, audible alarms, animal calls, etc). Indeed, V. Vemuri summarized the situation well.

". . .we usually recognize a familiar face in only about 200 milliseconds. The human eye can adjust to light intensity levels over 7 orders of magnitude. No man-made image processing system can come even close to this performance. It is remarkable that this performance is obtained by a system whose individual components are larger, slower, and noisier than state-of-the-art electronic components." [Vemuri, 1988]

Pattern recognition usually involves making binary decisions on relatively complex inputs, i.e., given all of the pixel values of a picture, produce a 1 if there is a tree in the picture and a zero otherwise.

Other types of pattern recognition schemes are also possible, however. For instance, the Hopfield network is often used in pattern reconstruction. When a noisy input pattern is applied to a Hopfield network, the network will settle to the nearest stored pattern. Ideally, if a noisy picture of a tree is applied to a Hopfield network, the network will settle to a prototypical picture of a tree.

Still another type of pattern recognition is to simply form groups of "similar" patterns without regard for what those patterns are. This is frequently referred to as clustering. Clustering has the advantage that the neural network learns simply from observing its environment. The drawback is that there is no guarantee that the clusters will correspond to identifications that are useful for the problem at hand.

1.2.2 Continuous Function Approximation

When the term "function approximation" is used in conjunction with neural networks, it usually refers to networks which have continuous-valued (or at least many valued) inputs and outputs. These networks are trained to approximate continuous functions.

Function approximation may be used, for instance, in the control of a robotic arm. The inputs could be the current joint angles, and the desired hand position. The neural network would then approximate the voltages required to drive the motors controlling the joints.

Since pattern recognition is usually limited to binary functions (e.g., tree present-tree not present) and function approximation includes all functions (including binary functions), function approximation is a more challenging problem. All of the experiments discussed in this dissertation involved function approximation.

1.3 Fault Tolerance

The term "fault tolerance" refers to the ability of a system to remain functional in the presence of faults. This feature is often highly desirable in systems where malfunction could be monetarily costly, systems where failure could lead to injury or death, or systems which are very remote and therefore difficult or impossible to service.

Fault tolerance is also desirable in complex systems, since manufacturing yields may be improved by allowing some faults in the finished result.

1.3.1 Redundancy

Perhaps the key to fault tolerance is some kind of redundancy or spare capacity. Clearly, if every component of a system is absolutely essential for proper operation, then a fault in any component is sufficient to cause the system to malfunction.

A common form of redundancy used in traditional systems is called N-modular redundancy [Siewiorek, 1991]. In this type of system, N (≥ 3) identical subsystems all work independently to produce an output. The output which is produced by a majority of the subsystems is taken to be the correct output. Thus, for instance if N is five, the system will perform correctly provided that at least three subsystems are functioning correctly.

1.3.2 Graceful Degradation, Fail Safe/Fail Soft

Since neural networks do not produce an exact answer, even under fault-free conditions, and since any fault in the system will inevitably affect the output of the system [Carter, 1988], the notion of fault tolerance rests on the notions of "graceful degradation" and "fail safe". The two notions are strongly related.

With graceful degradation, one recognizes that faults will adversely affect the performance of the system. One then strives not to eliminate any loss of performance, but rather to minimize loss of performance. Ultimately, when the loss of performance is sufficient that the system fails, one desires that the failure be "safe". For instance, in the case of a robot arm moving through a trajectory, one would desire that a controller failure would cause the arm to move very little or not at all, as opposed to flailing about wildly.

1.3.3 Distribution of Calculation

Artificial neural networks ultimately perform calculations. They produce outputs which are functions of their inputs. Because many processing elements are usually available to contribute to the calculation, it is widely believed that the loss of a few processing elements would not seriously impact the network performance. Experiments do not support this belief, however.

Merely having a large number of processing elements is not sufficient to guarantee that the processing will be well distributed. In some cases, for instance, a small number of processing elements will undertake the bulk of the calculation. Thus, the loss of any of these processing elements will seriously impact the performance of the network. In other cases, processing elements may not contribute constructively to the solution. For instance, two processing elements may contribute large and opposite values, cancelling each other out. Their net contribution is zero, although the loss of either one will lead to a large change in the network's performance.

CHAPTER II

FAULT TOLERANCE OF ARTIFICIAL NEURAL NETWORKS

2.1 Prior work

Artificial neural networks are widely believed to be inherently fault tolerant. Indeed, one frequently encounters claims such as the following:

"Neural networks are extremely fault tolerant and degrade gracefully. They can learn from and make decisions based on incomplete data. Because knowledge is distributed throughout a system rather than residing in a single memory location, a percentage of the nodes can be inoperative without significantly changing the overall system behavior." [Nelson, 1991]

Unfortunately, for those interested in fault tolerance Nelson's statement is simply not true. Fault tolerance is *not* an inherent property of artificial neural networks. However, it has been found that fault tolerance can be improved by appropriate training methods [Séquin, 1990] and furthermore, it has been found that fault tolerant solutions often exhibit better generalization (i.e., better responses to novel inputs) [Kruschke, 1989, Neti, 1990].

Few researchers have worked on theoretical and/or abstract issues in the study of fault tolerance of artificial neural networks. For more extensive reviews of prior work the interested reader might consult the papers by Carter [Carter, 1988] and Bolt [Bolt, 1991]. These authors have attempted to formulate well structured frameworks and appropriate questions for guiding fault tolerance research.

All in all, the field of fault tolerance of artificial neural networks is relatively unexplored. Much work remains to be done before the fault tolerance of artificial neural networks is fully understood; however, it is hoped that this dissertation adds significantly to the body of knowledge in important areas such as quantifying fault tolerance, improving fault tolerance, and *a priori* prediction of network fault tolerance.

2.2 Formulating a Fault Tolerance Measure

In order to compare the fault tolerance of various networks it is necessary to develop a meaningful fault tolerance measure. This measure should be applicable to a wide range of networks and applications.

Essentially, the problem of formulating a fault tolerance measure consists of two parts. The first part is to formulate a reasonable error measure. The second is to formulate a suitable faulting method. Measuring fault tolerance then becomes an exercise in applying the faulting method and observing the effect on the measured error.[3] Ultimately, of course, it is desirable to use the measured fault tolerance data to develop hypotheses about fault tolerance which may be tested.

2.2.1 Choosing an Error Measure

Three types of approximation error measurements will be considered in this dissertation, viz., maximum deviation, RMS error, and a normalized error measurement called the approximation quality (AQ). The relative advantages and disadvantages of each will be discussed.

2.2.1.1 Maximum Deviation

A maximum deviation error measure is one which characterizes the maximum absolute error over the input space. If a network is trained using a training set composed of N pairs of network inputs and desired network outputs (x_i, y_i), and the network produces a function F(x), then the maximum deviation is defined in Equation 2.1.

$$\text{Max Deviation} = \max_{i=1\ldots N} |F(x_i) - y_i| \qquad (2.1)$$

Put another way, if the network has a maximum deviation ε, then the network output is guaranteed to fall within an envelope of $\pm\varepsilon$ about the intended response. On the downside,

[3]Of course, one could also characterize the fault tolerance of a network using a rigorous mathematical analysis. This approach is impractical, however, for all but very small networks and very simple problems.

however, this method of error measurement greatly penalizes outliers. For instance, a network which performs nearly perfectly at all but a single point, where it has an error of ε, is indistinguishable from a network which has an error ε at every point.

A maximum deviation error measure is suitable for networks which learn binary functions since one is not interested in the actual network output, but only if the output is above or below some threshold. When a network is learning continuous functions, however, the maximum deviation at a single point is usually much less important than the overall fit of the network output to the desired output. Moody classifies neural networks as regression or classification systems depending upon whether the network outputs are real-valued or categorical. [Moody, 1991]

2.2.1.2 RMS Error

The RMS error is often a more appropriate error measure in function approximation tasks. Rather than measuring the maximum deviation, the RMS error gives a measure of the overall fit of the network output to the function throughout the input space. The RMS error is defined in Equation 2.2.

$$\text{RMS Error} = \left[\frac{1}{N}\sum_{i=1}^{N}(F(x_i) - y_i)^2\right]^{\frac{1}{2}} \qquad (2.2)$$

2.2.1.3 Approximation Quality

One problem inherent in both the maximum deviation and RMS error measuring schemes is that they don't account for differences in scale of the learned function. Thus, one must be extremely careful in comparing networks trained to the same approximation error on functions with large scale differences.

It is important to somehow normalize the RMS error with respect to the RMS value of the function to be learned. It is for this reason that the following error measure is proposed which is called AQ (standing for approximation quality).

$$AQ = \frac{\text{Function RMS}}{\text{Function RMS} + \text{RMS error}} \qquad (2.3)$$

This dimensionless measure is always between zero and one with the value one corresponding to no error and the value zero corresponding to infinite error. Furthermore, for a value of 0.5, the function RMS and the RMS error are equal.

2.2.2 Choosing a Faulting Method

Once an appropriate error measure has been decided upon, the next step is to choose an appropriate faulting method. There are a number of possibilities in choosing a faulting method. One might consider the average degradation or the worst case degradation. Additionally, one may consider single faults or multiple faults.

2.2.2.1 Average or Typical Single Fault

An easy faulting method is simply to apply single faults to the network. One could insert each possible fault and average the resulting approximation errors, or if one assumes that any fault is equally likely, one could average network performance results for a large number of randomly chosen faults rather than exhaustively testing all possible faults.

This faulting method is the simplest computationally. However, it does not provide a strict error bound, but only gives an expected error. Furthermore, this measure can artificially be improved for any network by merely adding "free" parameters which do not contribute to the network output in any way. The loss of any of these parameters does not degrade network performance and thus the overall average improves, despite the fact that the network is fundamentally unchanged.

2.2.2.2 Most Damaging Single Fault

A method which is perhaps a bit more telling is to measure the largest degradation caused by a single fault. This is approximately of the same computational complexity as the average single fault method, but has the advantage of providing a worst-case bound on the network performance degradation. Using the largest degradation rather than the average degradation obvicusly penalizes networks which have a small number of highly critical parameters. However, from a fault tolerance standpoint this is a perfectly reasonable property.

2.2.2.3 Multiple Faults

When extending the faulting method beyond single faults, the computational complexity of measuring fault tolerance begins to grow at a staggering rate. Since it is impractical to assume that a neural network will develop a maximum of one fault throughout its entire operating lifetime, it is important that some means be found to address the analysis of multiple faults in a network.

2.2.2.3.1 Combinatorial Growth of Possibilities

The number of ways that one can get N weight faults of a single type in a network having W weights is ${}_{W}C_{N}$, where:

$${}_{W}C_{N} = \frac{W!}{N!(W-N)!} \qquad (2.4)$$

Unfortunately, this expression grows very rapidly with N even for modest W values. For instance, Figure 2.1 shows the combinatorial growth in the number of weight faults for a network having fifty weights.

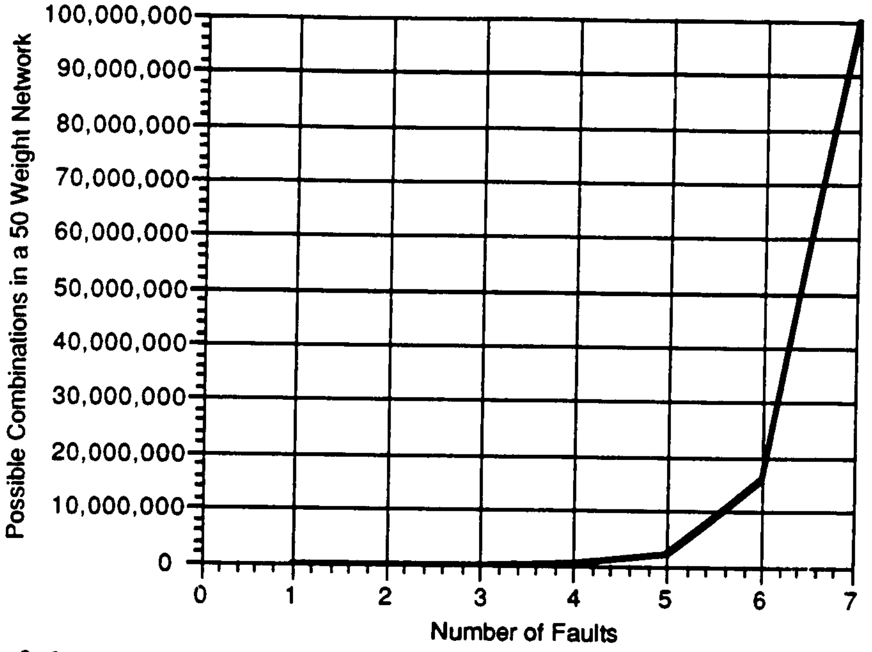

Figure 2.1 A graph showing the number of combinations of multiple weight faults (up to seven) for a network having fifty network weights.

As can be seen, evaluating even a modest number of weight faults can rapidly lead to an exorbitant number of calculations. Of course, if one has a larger network, the growth is even faster! This leads to a dilemma. One can only characterize the most damaging multiple weight fault of a given type by evaluating all of the possible multiple weight faults of that type, however, evaluating all of the combinations is simply not feasible. Some type of trade off must be made to ease the computational burden.

2.2.2.3.2 Sampling

One method that may be employed is to randomly sample from the possible faults of a single category (such as triple weight faults), and average the resulting network performances. Assuming that one samples a sufficient number of times, one should get a reasonable prediction of the typical behavior. One could also attempt to predict the worst case behavior by sampling from each category a large number of times and simply taking the worst performance.

There are two drawbacks to sampling. The first is that one cannot be sure that the samples that one takes are typical of the entire category. The second is that it does not avoid the problem of combinatorial growth if one samples a large percentage of each category.

The two drawbacks are related in opposing ways. In order to be more confident that one has captured typical behavior for the category, one must increase the number of samples taken. However, to ease the computational burden, one must decrease the number of samples taken. Thus, there is a trade off between confidence level and computational complexity.

2.2.2.3.3 A Worst Case Path Approach

A second method for evaluating multiple faults avoids combinatorial growth entirely. This method is a worst case path approach. The assumptions made are that faults occur one at a time, and at each step the fault causing the largest incremental performance degradation is selected. For instance, first one evaluates all single weight faults to find the weight which causes the maximum performance degradation. This weight is then constrained to be one of the weights used in subsequent multiple fault evaluations.

This method has the advantage of eliminating the combinatorial growth in evaluating multiple faults. Additionally, it finds a worst case failure path assuming that faults occur one at a time. The disadvantage is that one cannot be sure that there are not particular multiple fault combinations which cause larger degradation than the worst case path. For instance, there may exist three weights which when lost together cause a larger performance degradation than the loss of the first three weights in the worst case path. Nevertheless, the worst case path approach should give a reasonable approximation to worst case performance and is much less computationally expensive, being of order N^2. This method is used in subsequent chapters to compare the fault tolerance of different networks.

CHAPTER III

FAULT TOLERANCE OF THE MULTILAYER PERCEPTRON

3.1 Experiment Definition

The experiments described in this chapter used single input, single output networks containing a single hidden layer containing one hundred processing elements. The sine function was used as the model for the desired network response; that is, the network was presented with a value X and was to produce the value sin(X) as its output. The network was trained on the interval from -6 to 6. Thirty evenly spaced training points were used. The networks were trained for 100,000 epoches. The networks were trained using several different training methods. These are discussed below. The goal of the experiments was to see if particular weight solutions obtained by different training methods have any difference in fault tolerance, and also to compare the fault tolerance of different networks.

3.2 Training Methods

Three different training algorithms were investigated for the multilayer perceptron. These were standard backpropagation, backpropagation with momentum, and backpropagation with a flexible learning rate.

3.2.1 Backpropagation

Standard backpropagation [Parker, 1985, Rumelhart, 1986, Werbos, 1974] is simply a means of performing gradient descent in the weight space of the network. After each presentation of the training set, the weights are adjusted by a small amount in the direction which causes the maximum reduction in approximation error. This process is repeated a number of times until the desired degree of accuracy is obtained, or until the network no longer shows improvement.

One drawback of standard backpropagation is in choosing the learning rate (the "small amount" by which the weights are changed). Choosing a learning rate which is too small can lead to very slow convergence. Choosing a learning rate which is too large can cause excessive deviation from the true gradient and lead to undesirable learning behavior, such as failure to converge.

3.2.2 Backpropagation with Momentum

One common extension to backpropagation is to use a "momentum" term during weight update. Thus, the current weight update is composed of two parts: the learning rate times the gradient direction, and a momentum factor (less than one) times the last weight update. It has been shown that the momentum term effectively raises the learning rate when consecutive gradient calculations point in nearly the same direction. [Watrous, 1987]

3.2.3 Backpropagation with Flexible Learning Rate

Another learning method that was explored was backpropagation with a flexible learning rate. For the method used, the gradient direction is computed in exactly the same manner as standard backpropagation. Two steps of different length are then taken in the gradient direction. After each step the error is evaluated (but a new gradient is not computed). This gives three error values for three different step sizes (the initial error corresponds to a step size of zero). A parabola is fitted to these points, and the step size corresponding to the minimum of the parabola is calculated. This step is tried, and if it is the lowest error obtained then it is used. Otherwise, the non-zero step which caused the lowest error is used.

Although the algorithm requires a number of passes through the training set for each weight update, it only requires a single gradient calculation per weight update. The rationale is that weight updates are fairly inexpensive and that one may more than make up for the additional calculations by faster convergence. That is, although each epoch will certainly take longer, the network should converge in fewer epoches.

3.3 Experiment Results

A number of runs using different random initial weights and biases were made using each of the training methods. In order to make accurate comparisons, the networks for each run were initialized in the same way.

3.3.1 Learning Performance

Figures 3.1-3.3 show typical plots of RMS error versus epoch number. For the sake of comparison these graphs have been clipped at an RMS error of 1.0.

As can be seen, backpropagation with momentum outperforms the other methods in terms of asymptotic approximation error. Backpropagation with flexible learning rate outperforms the other two methods when the error is high, but ultimately loses out to backpropagation with momentum.

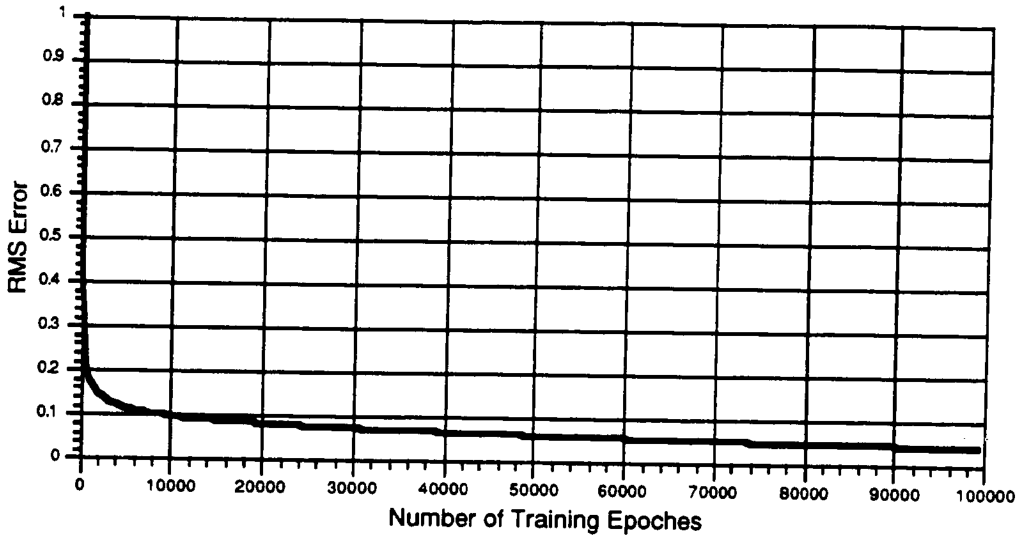

Figure 3.1 Typical graph showing the RMS error versus training epoch for the multilayer perceptron trained using standard backpropagation.

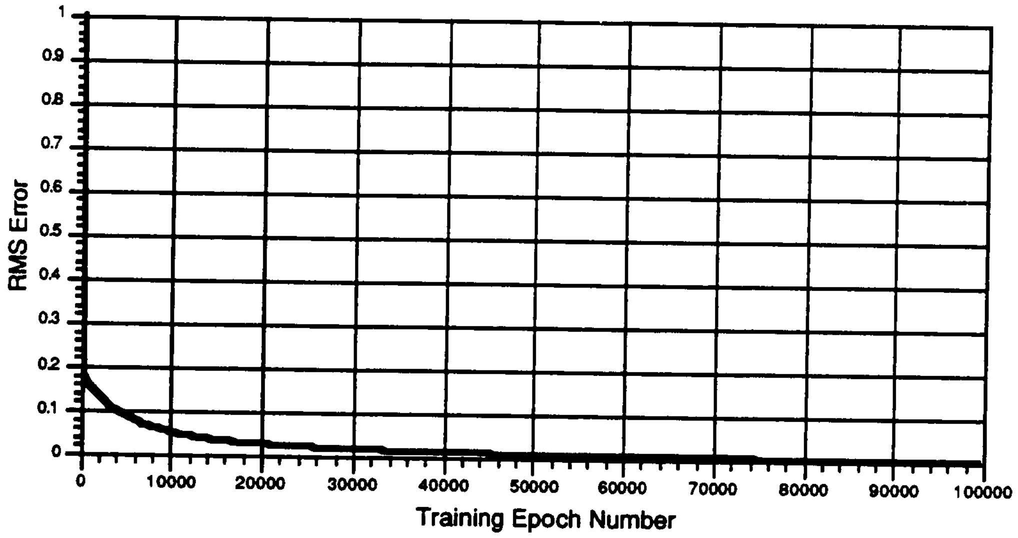

Figure 3.2 Typical graph showing the RMS error versus training epoch for the multilayer perceptron trained using backpropagation with momentum.

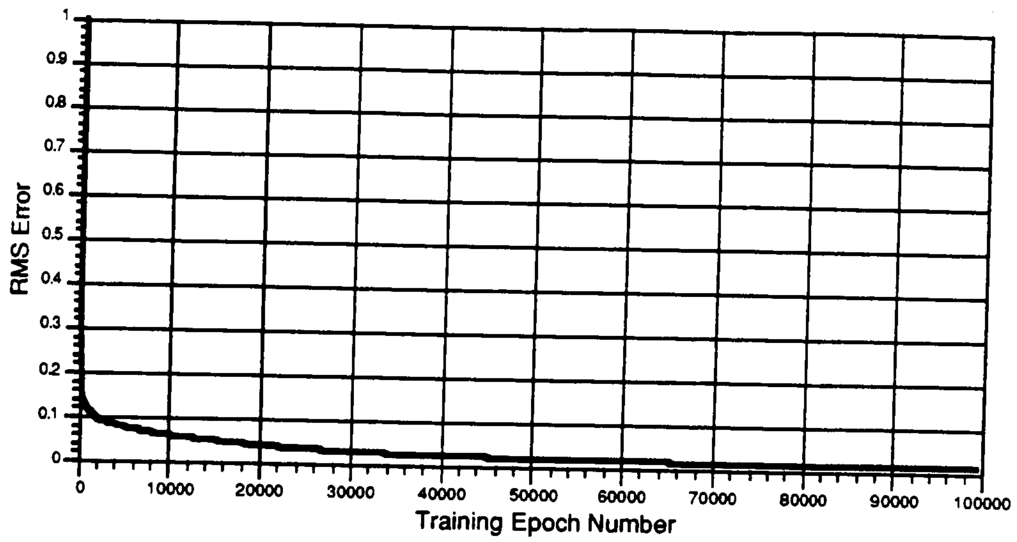

Figure 3.3 Typical graph showing the RMS error versus training epoch for the multilayer perceptron trained using backpropagation with flexible learning rate.

3.3.2 Single Weight Fault Tolerance

The following graphs (Figures 3.4-3.6) show the effect on RMS error of each of the single weight faults that can occur in the network. To generate the graphs, each network weight was set to zero and the change in RMS error was measured. The horizontal axis represents the value of the weight and the vertical axis represents the change in RMS error that the loss of the single weight caused. There are several important features to note. The first is that there is a relatively large number of critical weights. Since the function to be learned was the sine function, the function's RMS value was 0.707. However, there are a number of weights for which the loss of any single one will cause the RMS error to increase by more than 0.707.

As can be seen, the three different training methods produce different results. Backpropagation with momentum produces slightly better results than the other two methods in the sense that the worst case single weight fault yields an RMS error increase of approximately 1.4 versus 2 and 2.8 for the other training methods. Nevertheless, the loss of the single most critical weight in the network still produces a rise in RMS approximation error that is twice the function RMS value.

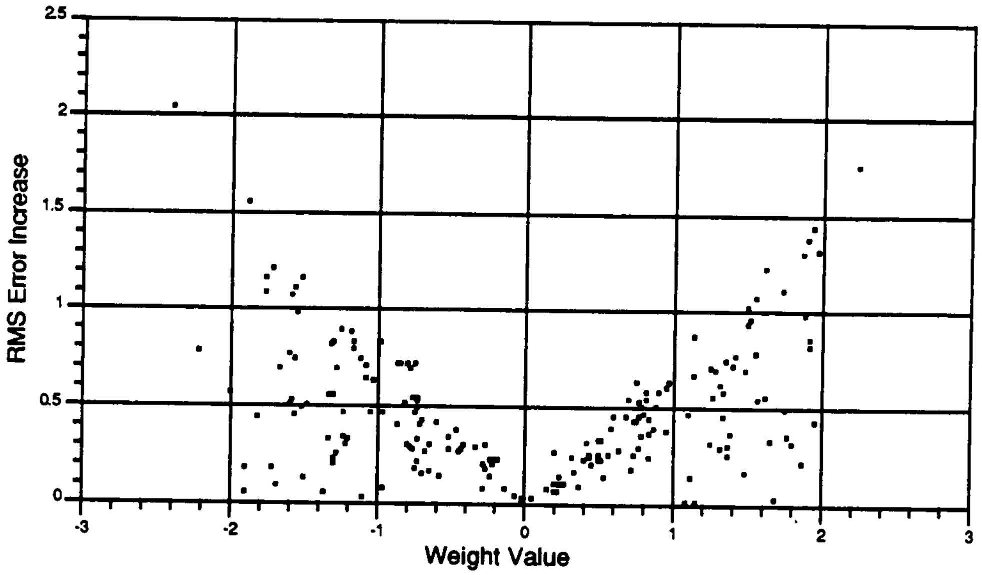

Figure 3.4 Graph showing the effect on RMS error of the loss of each single weight in the multilayer perceptron trained using ordinary backpropagation. The change in error is plotted as a function of the value of the weight.

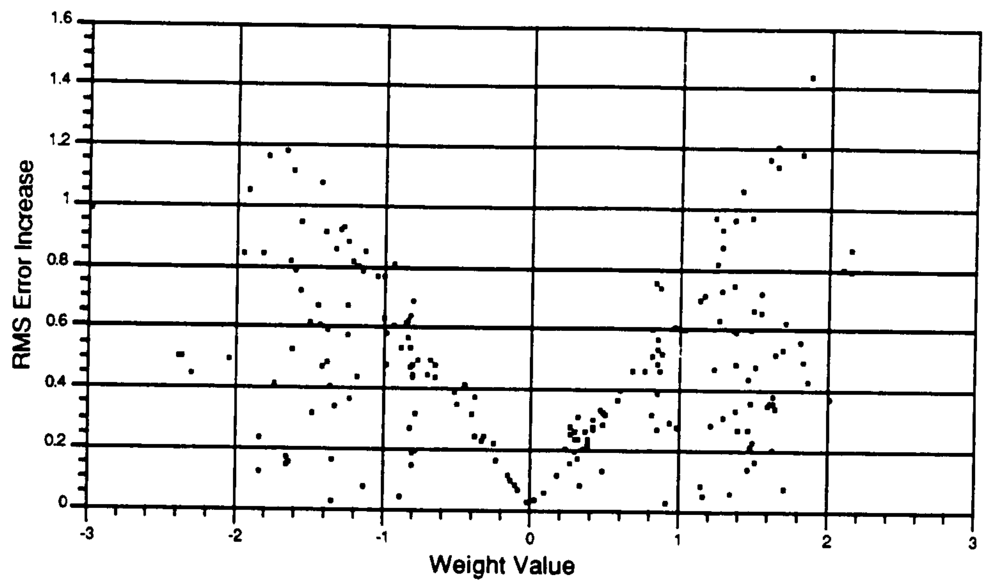

Figure 3.5 Graph showing the effect on RMS error of the loss of each single weight in the multilayer perceptron trained using backpropagation with momentum. The change in error is plotted as a function of the value of the weight.

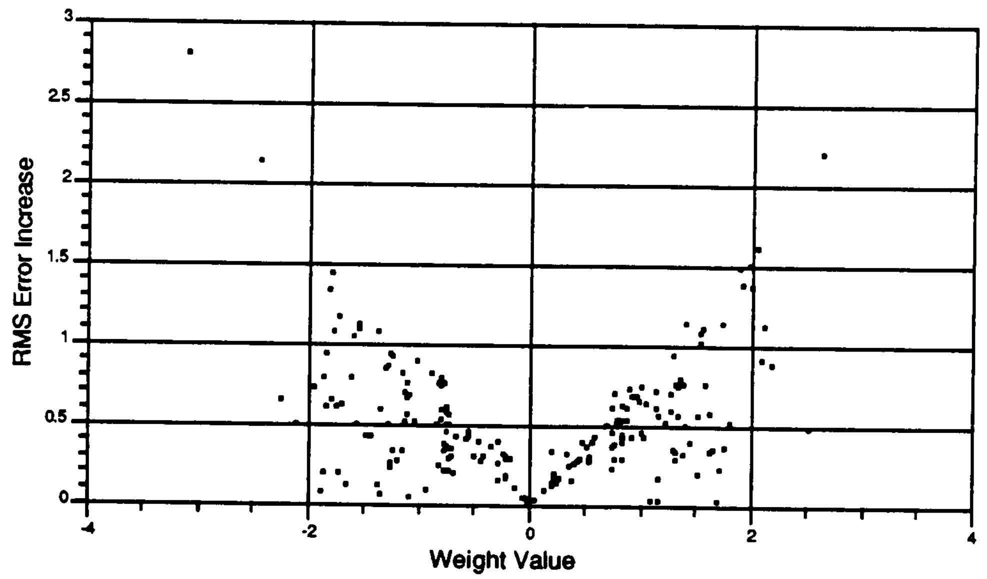

Figure 3.6 Graph showing the effect on RMS error of the loss of each single weight in the multilayer perceptron trained using backpropagation with flexible learning rate. The change in error is plotted as a function of the value of the weight.

3.3.3 Worst Case Path Performance

Figures 3.7-3.9 show typical worst case path weight loss performance for each of the three training methods. There are very few appreciable differences in the fault tolerance of the networks produced by any of the training methods. Again, backpropagation with momentum produces slightly better results than the other two training methods, but the fault tolerance is nevertheless very bad.

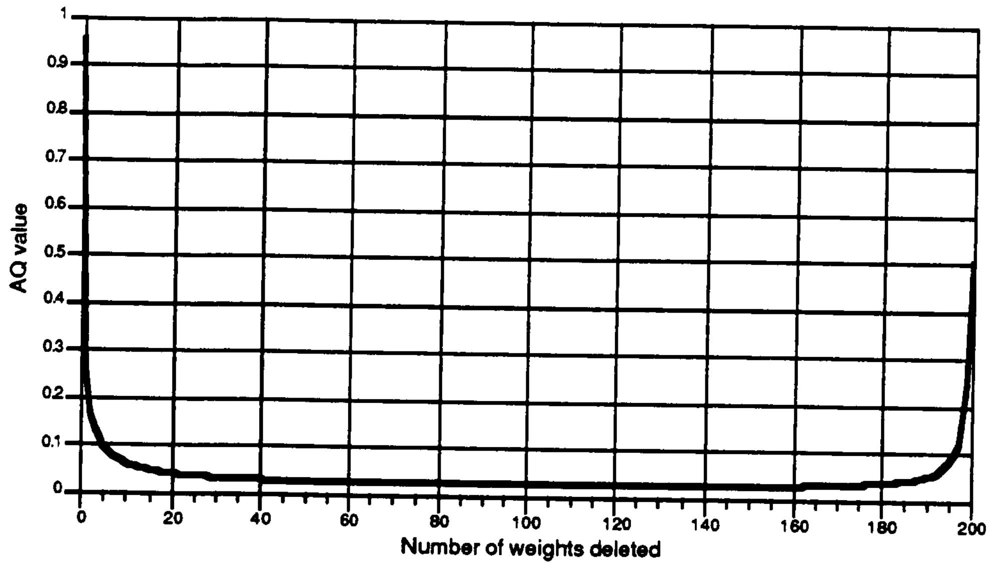

Figure 3.7 AQ value of the multilayer perceptron trained using standard backpropagation as weights in the worst case path are removed.

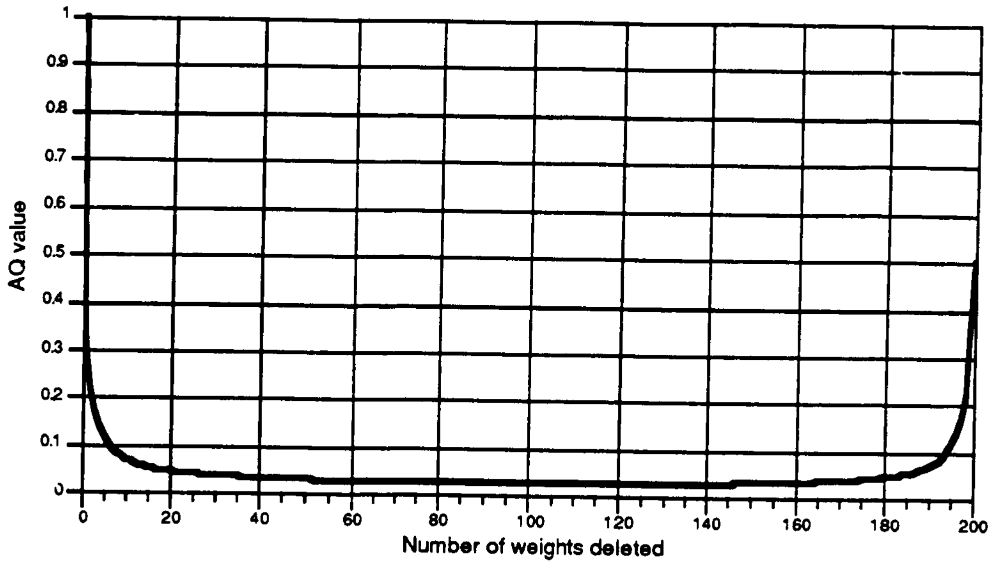

Figure 3.8 AQ value of the multilayer perceptron trained using backpropagation with momentum as weights in the worst case path are removed.

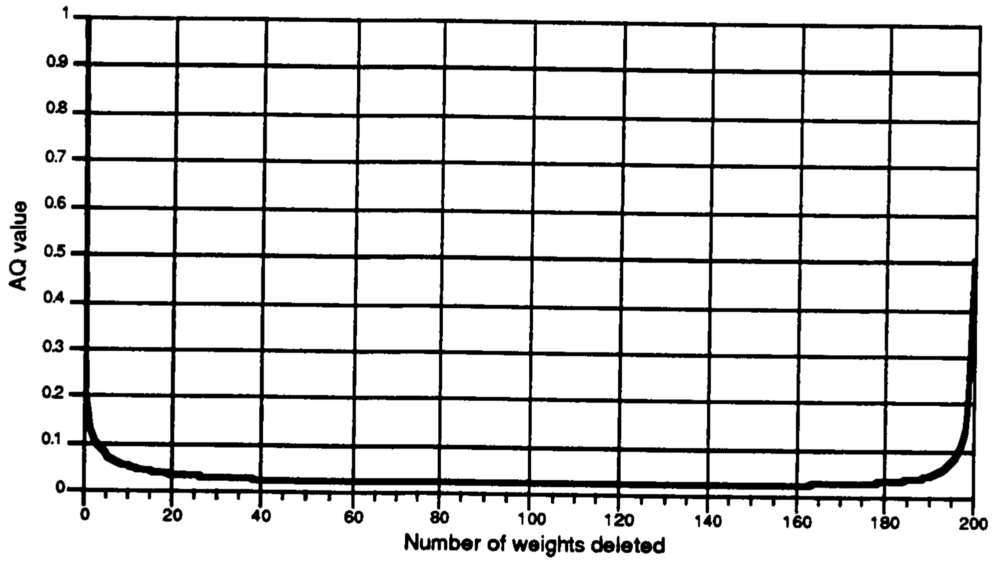

Figure 3.9 AQ value of the multilayer perceptron trained using backpropagation with flexible learning rate as weights in the worst case path are removed.

It is important to note two very striking features of these graphs. The first is that even in a network containing one hundred processing elements, the loss of the single most critical weight is sufficient to cause the RMS error to increase to a value much higher than the RMS value of the learned function. Thus, losing the single most critical weight is sufficient to render the network useless. The second feature to note is that as one follows the worst case fault path, the network error can increase by many times the learned function RMS (for instance, an AQ value of 0.05 corresponds to the RMS error being about 20 times as large as the function RMS). Thus, with the loss of only a few weights the network becomes "actively bad", producing errors well in excess of those which would result from having no network at all!

3.3.4 Partial Fault Tolerance

One important issue to address, particularly if one intends to implement artificial neural networks in hardware, is the effect of a small error in a network parameter (as opposed to the total loss of a parameter). In this dissertation, the the ability of a network to tolerate a small error (plus or minus ten percent) in one of its weights shall be referred to as the partial fault tolerance of the network. In order to study the partial fault tolerance of the multilayer perceptron, each weight in the trained network was increased and decreased by ten percent of its original value and the effect on RMS error was plotted as a function of the weight's original value. These plots are shown in Figures 3.10 through 3.12. In these plots the '+' symbol corresponds to the addition of ten percent of the weight value and the '-' symbol corresponds to the subtraction of ten percent of the weight value.

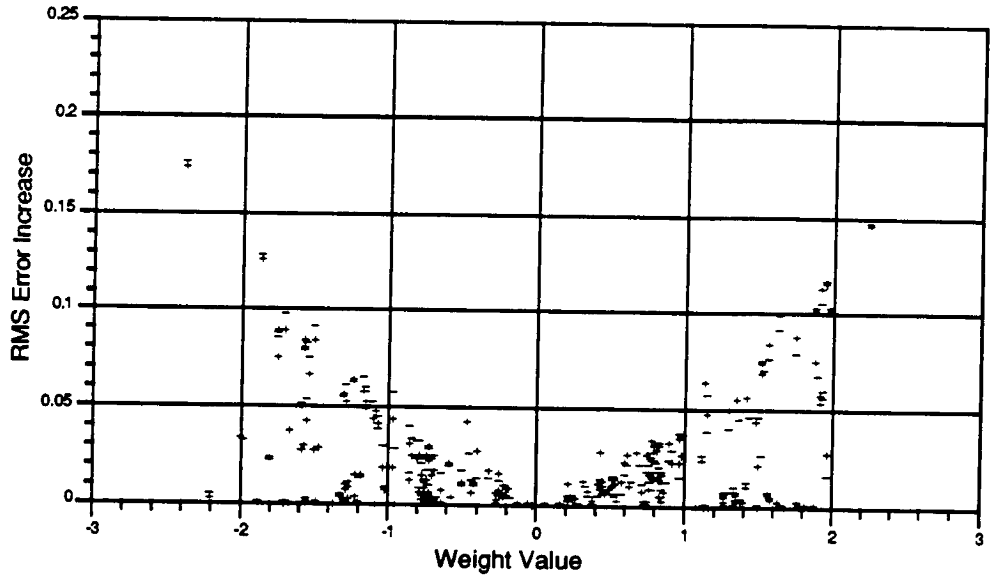

Figure 3.10 Effect on RMS error of a ±10% change of each single weight (individually) in the multilayer perceptron trained using ordinary backpropagation. The error increase is plotted as a function of the original value of the weight.

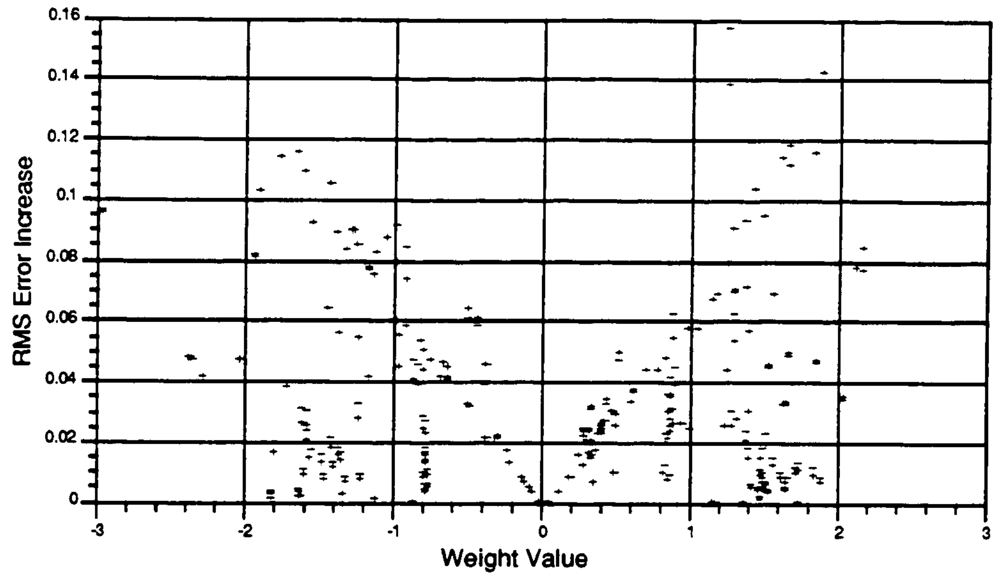

Figure 3.11 Effect on RMS error of a ±10% change of each single weight (individually) in the multilayer perceptron trained using backpropagation with momentum. The error increase is plotted as a function of the original value of the weight.

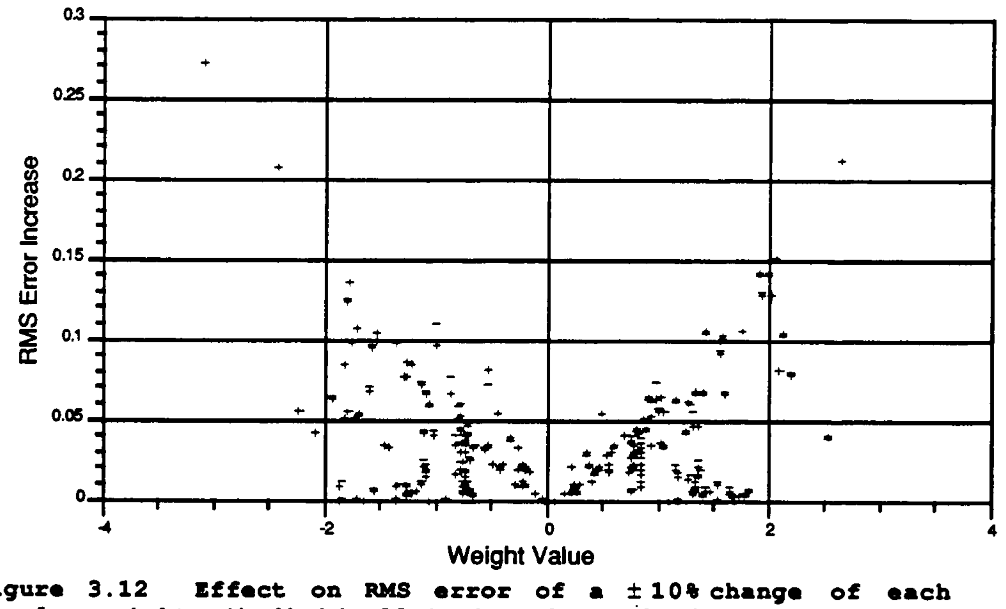

Figure 3.12 Effect on RMS error of a ±10% change of each single weight (individually) in the multilayer perceptron trained using backpropagation with flexible learning rate. The error increase is plotted as a function of the original value of the weight.

Needless to say, the graphs shown in Figures 3.10, 3.11, and 3.12 are very similar in shape to the plots shown in Figures 3.4, 3.5, and 3.6, respectively. Interestingly, a loss or gain of ten percent in the value of a weight causes approximately the same increase in RMS error, and this increase is approximately ten percent of the increase caused by the total loss of the weight.

3.4 The Effect of Network Pruning

Multilayer perceptrons are well suited to a variety of tasks, but have the drawback that they tend to learn very slowly, and thus require a great deal of training. In an effort to speed training, a technique which has shown considerable promise involves beginning with a network larger than needed, and as training proceeds, one gradually reduces the size of the network by removing nodes which do not contribute significantly. The method by which the size of the network is reduced is frequently called pruning. There are a variety of techniques for determining when and where to prune such as skeletonization [Mozer, 1989], weight-elimination [Rumelhart, 1988] and optimal brain damage [LeCun, 1990].

Work was done [Segee, 1990] to assess the fault tolerance of multilayer perceptrons trained using pruning and to compare this to the fault tolerance of multilayer perceptrons of the same original size trained without pruning (the initial weights were kept identical).

Only single weight faults were considered, and the error criterion used was RMS error. The results of the investigation were that the pruned network was neither significantly more fault tolerant nor significantly less fault tolerant than the unpruned network. Figures 3.13 and 3.14 show typical results.

In order to generate these graphs a network with one hundred processing elements arranged in a single hidden layer was trained for 30,000 training epoches. The training points were samples from a sine function. Another initially identical network was trained using the same training set. In this network, however, every five hundred epoches the node which was determined to be the least relevant to the calculation was permanently removed from the network. Thus, the network was pruned form its original size of one hundred nodes to a final size of thirty nodes.

Figure 3.13 shows the effect of each possible single weight fault in the unpruned multilayer perceptron. Figure 3.14 shows the corresponding graph for the pruned network.

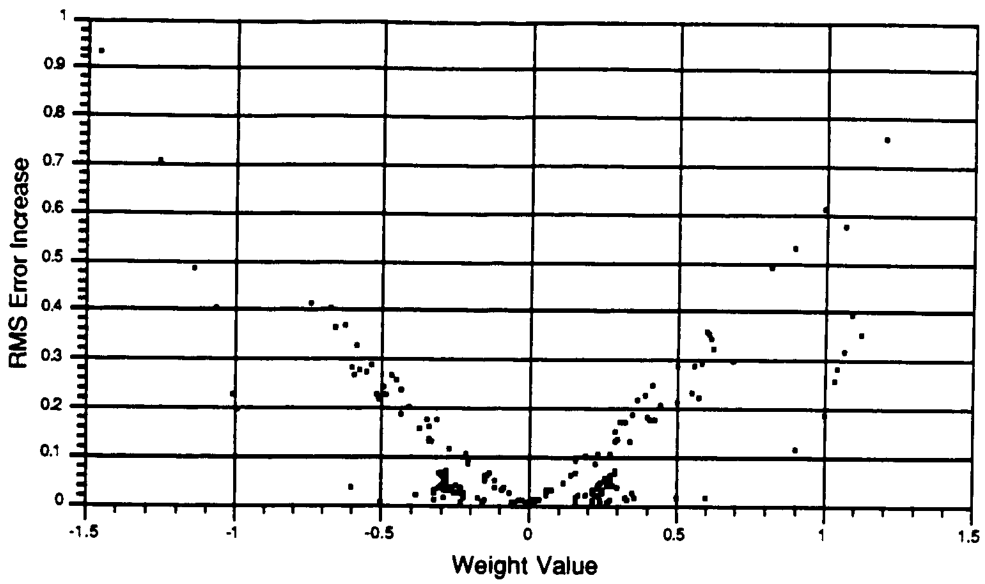

Figure 3.13 Graph showing the effect on RMS error of the loss of each single weight in the multilayer perceptron trained using ordinary backpropagation. The change in error is plotted as a function of the value of the weight. The initial weights for this network were the same as those for the network represented by Figure 3.14.

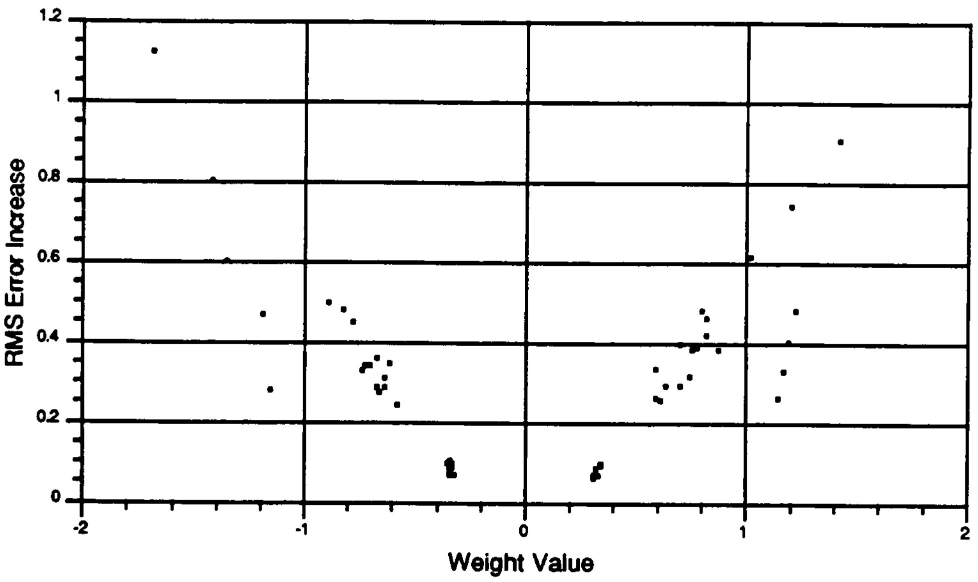

Figure 3.14 Graph showing the effect on RMS error of the loss of each single weight in the multilayer perceptron trained using backpropagation with pruning. The change in error is plotted as a function of the value of the weight. The initial weights for this network were the same as those for the network represented by Figure 3.13.

While it is important to know that the fault tolerance is not significantly reduced by pruning during training, the poor fault tolerance performance of the unpruned multilayer perceptron limits the benefit that one may derive from these findings. Perhaps most importantly, these findings clearly demonstrate that the training algorithm used does not make full use of available network parameters. That is, a fairly small number of parameters have any significant impact on the quality of the function approximation. Removing parameters which have a very small impact on performance does little to alter the properties of the network. Thus, pruning does not significantly worsen the fault tolerance, however, the fault tolerance could be improved if the weights removed by pruning were instead used in a productive manner.

CHAPTER IV

FAULT TOLERANCE OF A RADIAL BASIS FUNCTION NETWORK

4.1 Experiment Definition

The experiment described in the previous chapter was repeated using a radial basis function network. The training again consisted of 30 evenly spaced samples of the sine function from the domain [-6, 6]. Units having a Gaussian activation function were used. Each Gaussian unit had the activation function shown below.

$$f_i(x) = e^{-\frac{(x - c_i)^2}{\sigma_i^2}} \qquad (4.1)$$

Here, the subscript i denotes the unit number. Each unit has a center parameter c_i and a width parameter σ_i.

The network output consisted of a weighted sum of the individual unit outputs as shown below.

$$F(x) = \sum_{i=1}^{N} w_i f_i(x) \tag{4.2}$$

Two different network configurations were tested: one had the same number of units as the multilayer perceptrons discussed in the previous section (viz., one hundred), and the other had the same number of weights (viz., two hundred).

4.2 Training Methods

Training a multilayer perceptron is relatively straightforward. This results from the two different types of parameters, weights and biases, being adjusted by the same learning rule. The biases can be thought of as being weights connected to a unit whose output is always unity.

The radial basis function network on the other hand, has three parameters, each of which is qualitatively distinct. These are the weights, the centers, and the widths. Each of these parameters is considered separately below.

4.2.1 Weights

If one uses gradient descent weight adjustment with a squared error objective function (show below), the weights of the radial basis function network are the easiest parameters to adjust.

The (pointwise) squared error function is given by:

$$E = (F(x) - F_{desired}(x))^2 \qquad (4.3)$$

Since the radial basis function network is composed of a single layer of processing elements, and the output is a weighted sum of the output of each of the processing elements, the gradient direction of the error with respect to each of the network weights is simply proportional to the magnitude of the output of the corresponding processing element. Mathematically, the gradient of the error with respect to any network weight W_i is given by:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial F(x)} \frac{\partial F}{\partial w_i} = 2\,(F(x) - F_{desired}(x))\; e^{-\frac{(x - c_i)^2}{\sigma_i^2}} \qquad (4.4)$$

But, since each processing element sees the same error, the gradient direction for any weight is simply proportional to:

$$\frac{\partial E}{\partial w_i} \propto e^{-\frac{(x - c_i)^2}{\sigma_i^2}} = \frac{\partial F}{\partial w_i} \qquad (4.5)$$

For the experiments discussed in this dissertation the network weights were initialized to zero, and were adjusted during learning by gradient descent. The weight update rule used is given by Equation 4.6:

$$\Delta w_i = -\beta\,(F(x) - F_{desired}(x))\,\frac{e^{-\frac{(x-c_i)^2}{\sigma_i^2}}}{\sum_i \left(e^{-\frac{(x-c_i)^2}{\sigma_i^2}}\right)^2} \qquad (4.6)$$

The value β is the learning rate. This weight update learning rule is set up so that if β is set to one, the error will go to zero with a single weight update.

4.2.2 Centers

Adjustment of the center locations during training is somewhat more of a problem than adjustment of the weights. An obvious choice would be to use gradient descent for the center locations in a similar manner to that used for the weights. Indeed, the gradient of the error with respect to a center c_i is given in Equation 4.7.

$$\frac{\partial E}{\partial c_i} = \frac{\partial E}{\partial F(x)}\frac{\partial F}{\partial c_i} = \frac{4\,(F(x) - F_{desired}(x))\,(x - c_i)\,w_i}{\sigma_i^2}\; e^{-\frac{(x-c_i)^2}{\sigma_i^2}} \qquad (4.7)$$

In practice, the method of gradient descent adjustment of the center parameters was found not to work well. The graphs show in Figure 4.1 and 4.2 illustrate the problem involved.

The graphs show the magnitude of $\frac{\partial F(x)}{\partial c_i}$ as a function of $(x-c_i)$ for a unity weight. Figure 4.1 shows the curve for a width parameter of 2 and Figure 4.2 shows the curve for a width parameter of 0.25.

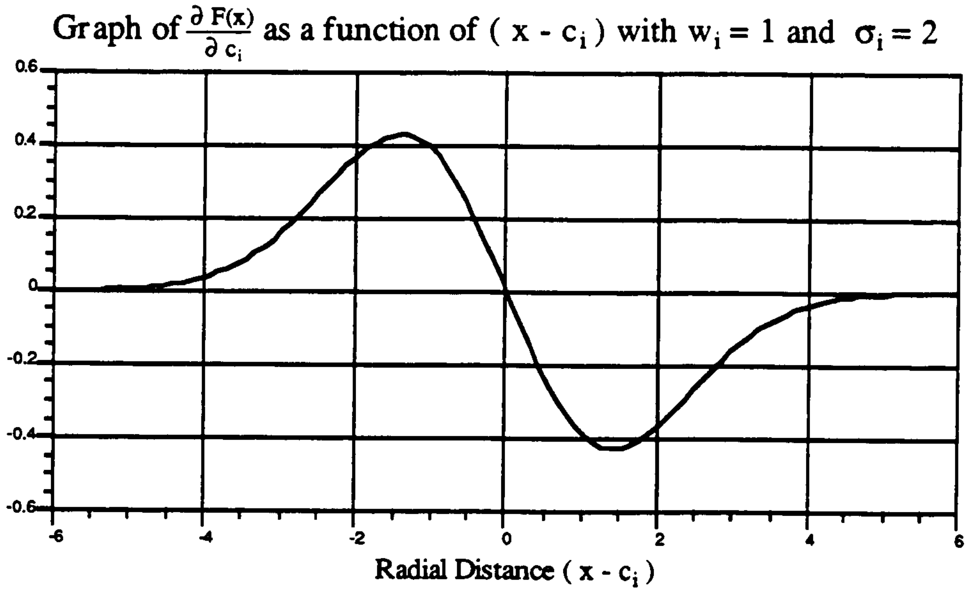

Figure 4.1 A graph showing the magnitude of $\frac{\partial F(x)}{\partial c_i}$ as a function of $(x-c_i)$ for a unity weight and width parameter of two.

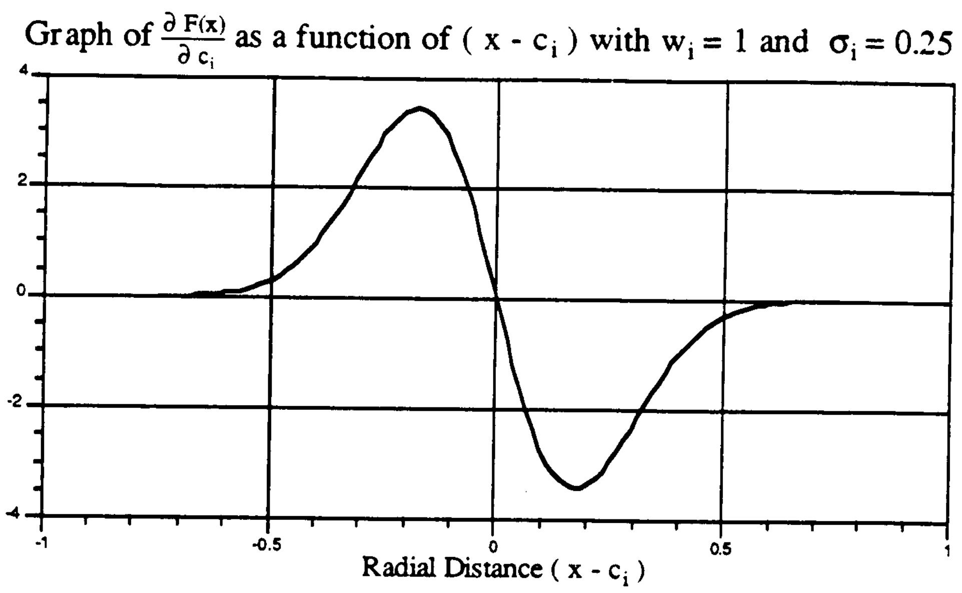

Figure 4.2 A graph showing the magnitude of $\frac{\partial F(x)}{\partial c_i}$ as a function of $(x-c_i)$ for a unity weight and width parameter of 0.25.

Two important observations can be made from these graphs. The first is that they go to zero very quickly as the magnitude of $(x-c_i)$ becomes large. Thus, if a processing element's center is not near any training data, it will never move. (Note that a similar statement can be made about processing elements with small weights, regardless of their proximity to training data.)

The second observation is that the magnitude of the partial derivative is highly dependent on the width parameter. This leads to a problem when choosing a step size to use. If one chooses a very small learning rate then the centers of the processing elements may never move, because of small weights and relatively large distances from the training data. If, on the other hand, one chooses a learning rate which causes reasonable changes in center locations, then one could end up making very large adjustments to the centers of processing elements which are relatively close to the training data. In fact, one could make so large an adjustment that these processing elements are no longer near any of the training data and therefore are lost.

A common heuristic method for adjusting the locations of the centers of the processing elements is a competitive, unsupervised form of clustering [Darken, 1990]. This method simply chooses the closest processing element at each training point and moves its center toward the current point. This method works best when there is a relatively small number of processing elements compared to the training set size and the training points are presented in a random order. Neither of these conditions are true for the experiments described in this dissertation. Thus, not surprisingly, this method did not work well.

For the experiments described in this dissertation the centers of the Gaussian units were evenly spaced across the range of the training data and were not adjusted during training.

4.2.3 Widths

For the sake of completeness, the gradient of the error with respect to the width parameter is given in Equation 4.8.

$$\frac{\partial E}{\partial \sigma_i} = \frac{\partial E}{\partial F(x)} \frac{\partial F}{\partial \sigma_i} = \frac{4\,(F(x) - F_{desired}(x))\,(x - c_i)^2\, w_i}{\sigma_i^3}\, e^{-\frac{(x - c_i)^2}{\sigma_i^2}} \qquad (4.8)$$

Virtually all of the problems present in performing gradient descent adjustment of the center parameters are equally problematic if one attempts to adjust the widths via gradient descent. Namely, the gradient is nearly zero at all points more than a moderate distance from the processing element, and near the processing element (particularly if the width is small) the gradient becomes very large. The effect on centers is to move them far from the training data. The effect on the widths is to shrink them to zero (or negative), causing all sorts of mathematical problems.

Unless stated otherwise, the width parameter value was chosen to be one and was not updated during training for the experiments described in this dissertation.

4.3 Experiment Results

The network weights were initialized to zero, and were adjusted during learning by gradient descent. Two different network configurations were tested: one had the same number of units as the multilayer perceptrons discussed in Chapter 3 (viz., one hundred), while the other had the same number of weights (viz., two hundred).

4.3.1 Learning Performance

Figure 4.3 shows a graph of RMS approximation error as a function of training epoch for the Gaussian radial basis function network with one hundred units. Figure 4.4 shows a graph of RMS error as a function of training epoch for the Gaussian radial basis function network having two hundred units.

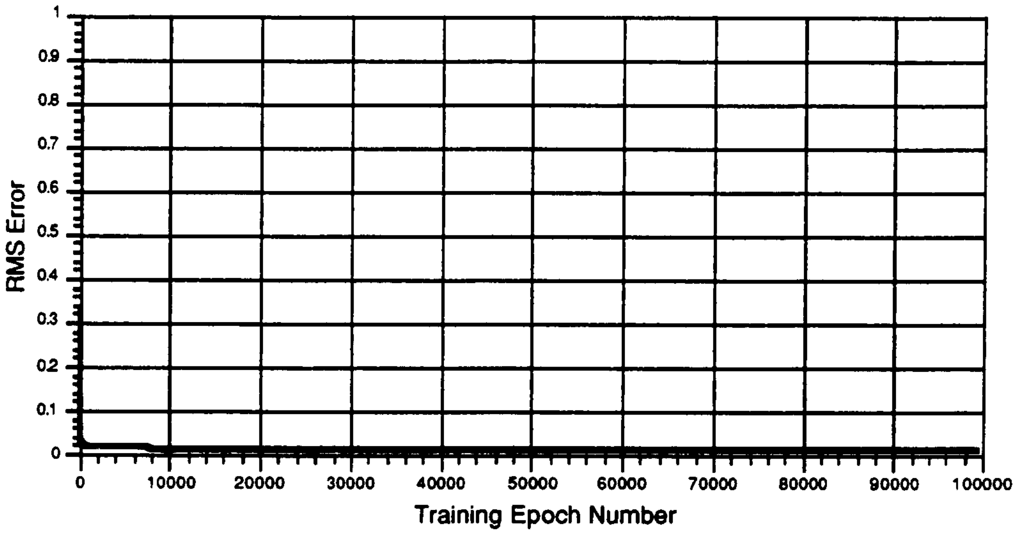

Figure 4.3 A graph showing the RMS error versus training epoch for a Gaussian radial basis function network with one hundred units uniformly spaced and with a unity width parameter.

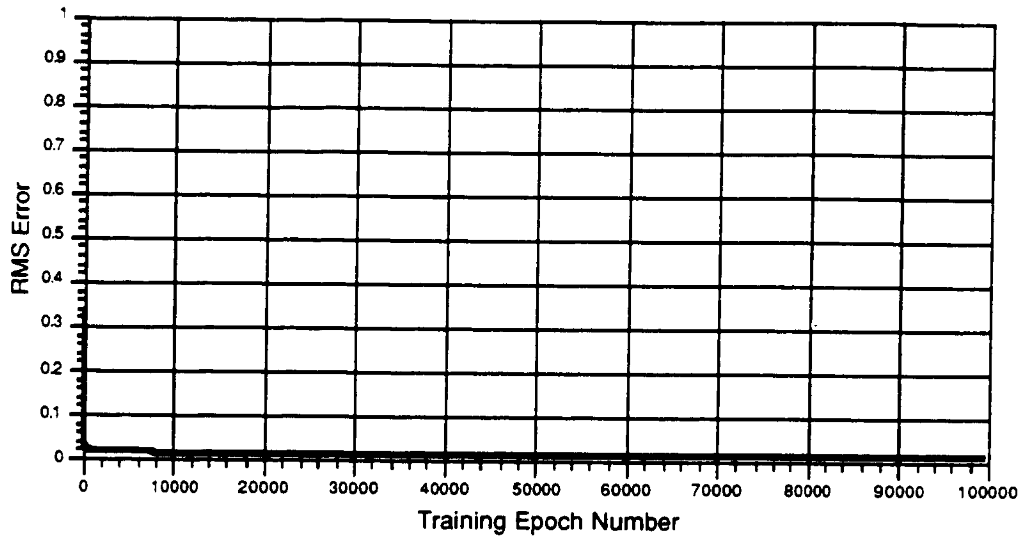

Figure 4.4: A graph showing the RMS error versus training epoch for a Gaussian radial basis function network with two hundred units uniformly spaced and with a unity width parameter.

In terms of final RMS approximation error the Gaussian radial basis function network performed approximately as well as backpropagation with a flexible learning rate, i.e., slightly better than standard backpropagation and slightly worse than backpropagation with momentum. This network converged to a low error much more quickly than a multilayer perceptron did, however. In fact, the RMS error changed very little after about 500 epoches or so. By comparison, the multilayer perceptron took about 50,000 epoches to reach a similar state. The Gaussian radial basis function network having two hundred units converges to a slightly lower error than the one hundred unit network, but not significantly so.

4.3.2 Single Weight Fault Tolerance

In terms of fault tolerance, however, the Gaussian radial basis function networks far outperform the multilayer perceptron both in terms of single weight faults and the worst case path (discussed in the next section). Figures 4.5 and 4.6 are the results of testing the loss of each network weight individually in the one hundred unit network and the two hundred unit network respectively.

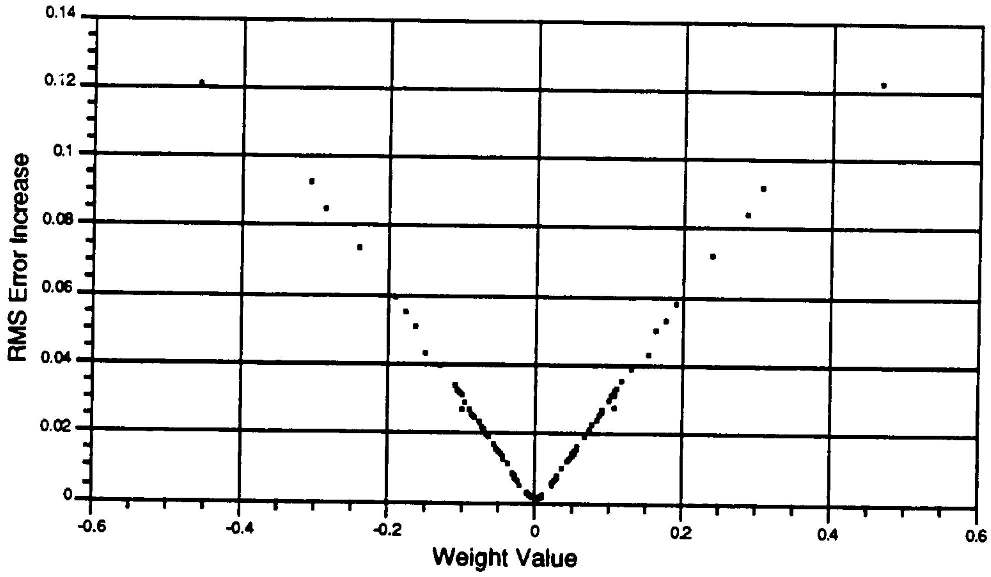

Figure 4.5 Effect on RMS error of the loss of each single weight in the Gaussian radial basis function network plotted as a function of the value of the weight lost. The network had one hundred processing elements uniformly spaced in the input domain; each one had a unity width parameter.

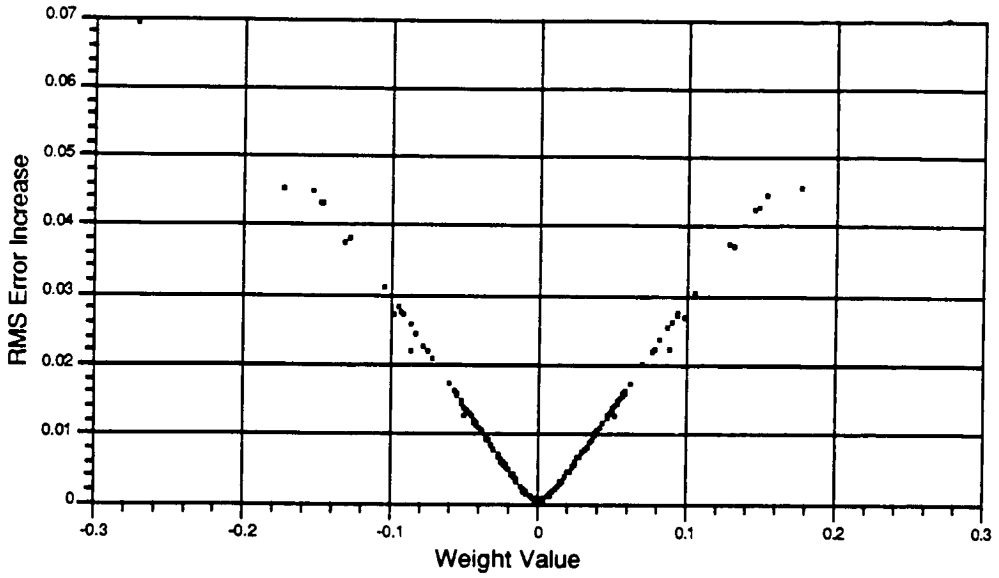

Figure 4.6: Effect on RMS error of the loss of each single weight in the Gaussian radial basis function network plotted as a function of the value of the weight lost. The network had two hundred processing elements uniformly spaced in the input domain; each one had a unity width parameter.

As can be seen, the two graphs are similar in shape and very nearly linear with respect to weight magnitude. The network with two hundred units had weights approximately half as large as those present in the one hundred unit network, and consequently these weights had approximately half the impact on performance when lost.

4.3.3 Worst Case Path Performance

The graphs showing the worst case path performance (in terms of approximation quality) for the Gaussian radial basis function networks are shown below. Figure 4.7 shows the fault tolerance curve for the network with one hundred processing elements and Figure 4.8 shows the fault tolerance curve for the network with two hundred processing elements.

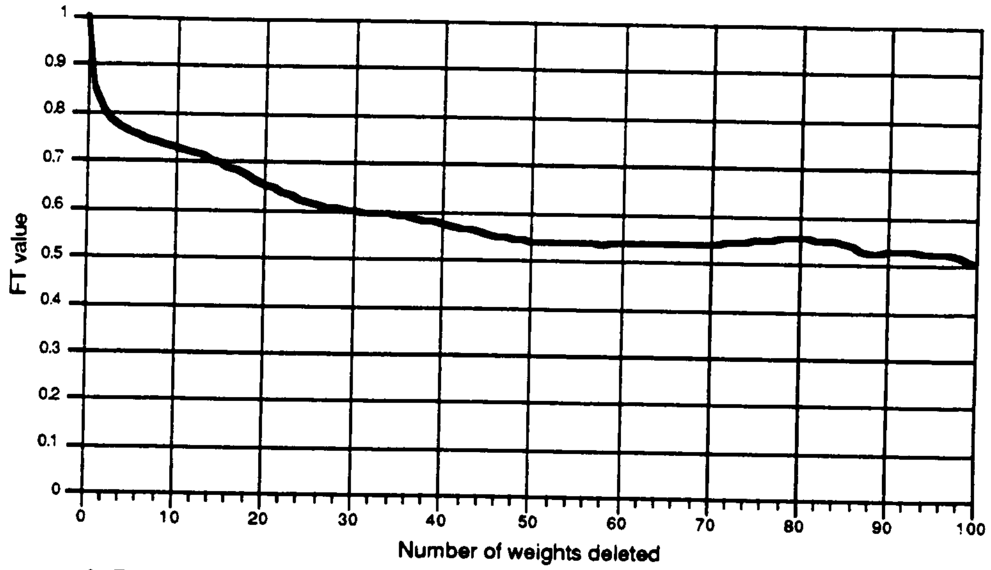

Figure 4.7 Effect on the AQ value of the Gaussian radial basis function network as weights in the worst case path are removed. The network had one hundred units uniformly spaced in the input domain; each one had a unity width parameter.

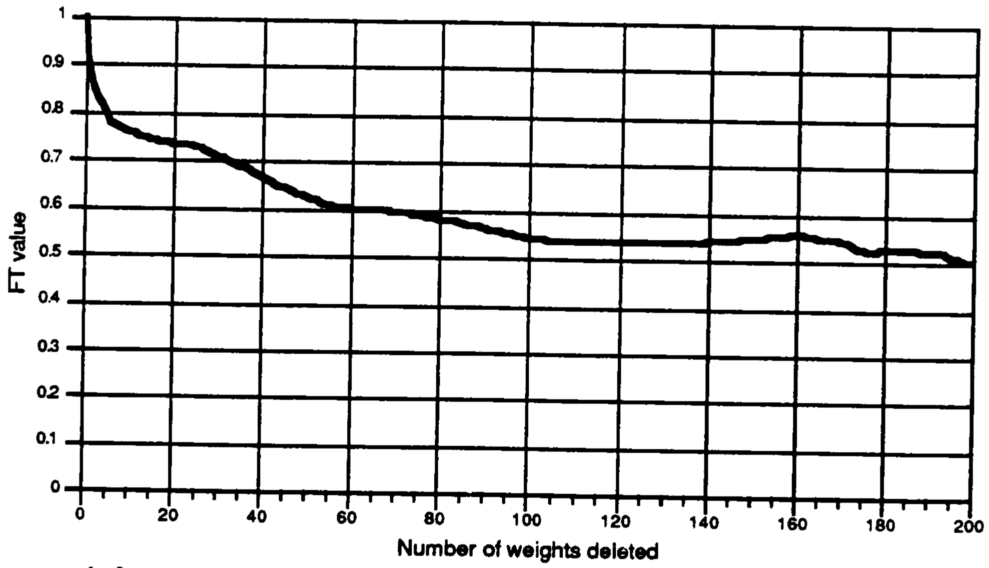

Figure 4.8: Effect on the AQ value of the Gaussian radial basis function network as weights in the worst case path are removed. The network had two hundred units uniformly spaced in the input domain; each one had a unity width parameter.

There are several very important things to observe from these graphs. The first is that the Gaussian radial basis function network is much more fault tolerant than the multilayer perceptron. As the worst case path is followed the AQ measure slowly approaches a value of 0.5. The AQ value never drops below 0.5, and thus the network always performs better than no network even in the face of an extreme number of faults. Remarkably, the shape of the two curves is virtually the same despite the difference in scale on the abscissa. Thus, doubling the number of units in the network, approximately doubles the number of weights in the worst case path that can be deleted before a given AQ level is reached. This has been tested for networks ranging from sixteen units to 512 units.

4.3.4 Partial Fault Tolerance

The effect on RMS approximation error when each weight in a Gaussian radial basis function network was changed by ±10% is shown in Figure 4.9. The '+' symbol represents the effect of adding ten percent to a weight, the '-' symbol represents the effect of subtracting ten percent from a weight.

As was the case with the multilayer perceptron, either adding ten percent or subtracting ten percent from a weight produces approximately the same increase in RMS approximation error. By comparing Figure 4.9 to Figure 4.5, one can observe that the RMS error increases are approximately ten percent (actually somewhat less) of the changes caused by the total loss of a weight.

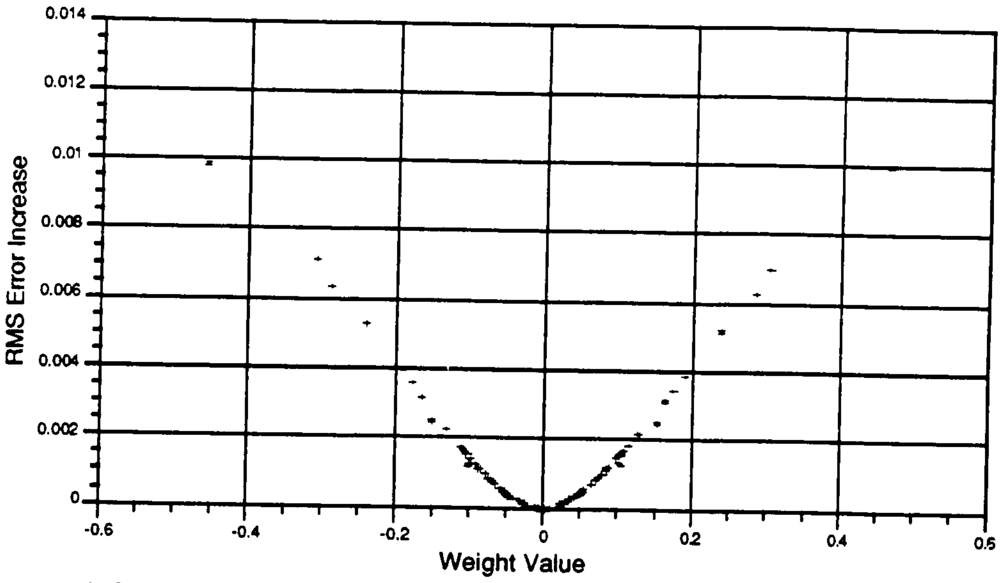

Figure 4.9 Effect on RMS error of a ± 10% change of each single weight (individually) in the Gaussian radial basis function network plotted as a function of the original value of the weight lost. The network had one hundred processing elements uniformly spaced in the input domain; each one had a unity width parameter.

A plot of the partial fault tolerance for the Gaussian radial basis function network with two hundred processing elements is shown in Figure 4.10. Its shape and behavior are substantially similar to the plot shown in Figure 4.9. Note that the scale of the ordinate and abscissa are approximately half of the values used for the one hundred unit network.

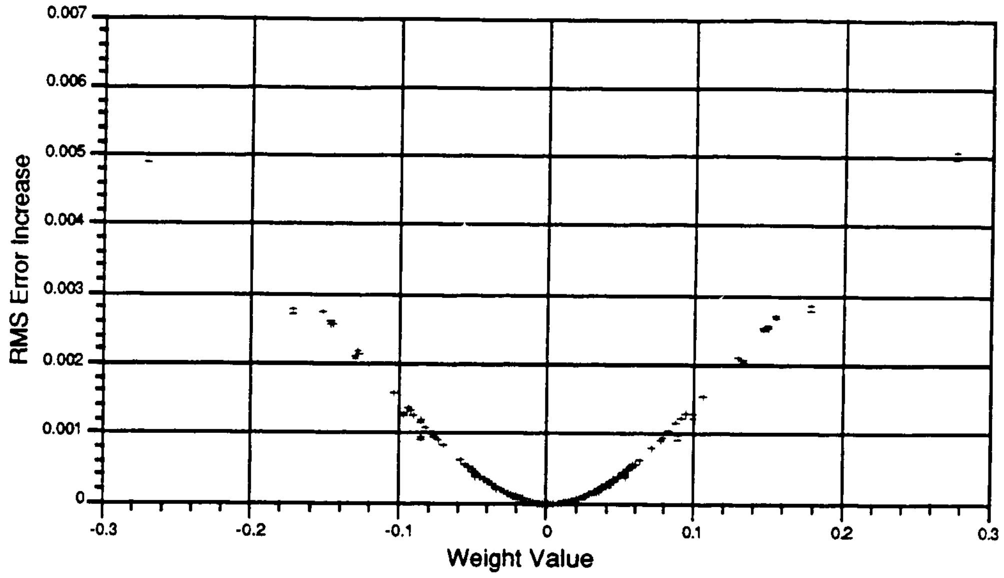

Figure 4.10: Effect on RMS error of a ±10% change of each single weight (individually) in the Gaussian radial basis function network plotted as a function of the value of the weight lost. The network had two hundred processing elements uniformly spaced in the input domain; each one had a unity width parameter.

It has been demonstrated that the Gaussian radial basis function network is more fault tolerant than the multilayer perceptron. It will be shown in Chapter 5 that this is due in large part to the choice of the basis function width parameter, i.e., a poor choice of width parameter can substantially reduce the fault tolerance of the network.

CHAPTER V

USING SPECTRAL TECHNIQUES FOR ANALYZING NEURAL NETWORKS

5.1 Function Approximation With Neural Networks

Function approximation using an artificial neural network is a process of function synthesis using a (potentially) non-orthogonal set of basis functions. The approximation consists of the weighted sum of the individual basis functions. "Learning", in the neural network usage, is the process of choosing the weights, and in some cases choosing the basis functions as well.

For the sake of simplicity, the discussions in this chapter will be limited to single-input, single-output systems, although the ideas extend naturally to multiple input networks. Additionally, any multiple-output network can be realized by multiple single-output networks.

A diagram for a single-input, single-output network is shown in Figure 5.1. The processing elements in Figure 5.1 are idealized, that is, any required parameters (such as biases, or center locations) are internal, links may or may not have adjustable weights, and the activation function may be any function.

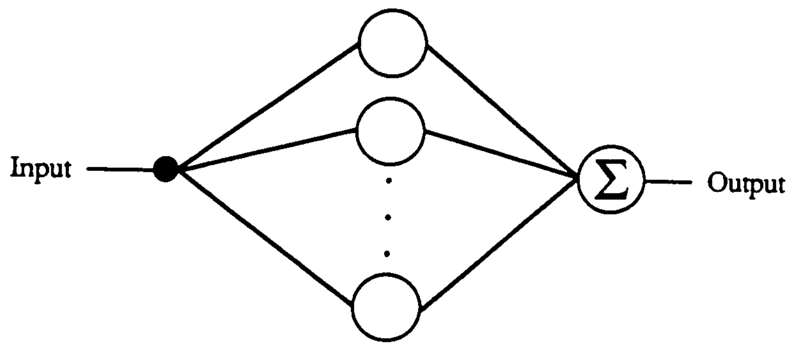

Figure 5.1 A single-input, single-output network.

A single layer of processing elements is shown in Figure 5.1. This is sufficient for the purposes of this chapter, since the methods described extend easily to multiple layers of processing elements. Indeed, for virtually any multilayer network, there is a realization in terms of a single layer of (potentially) different complex processing elements that create virtual basis functions in the input domain. The methods of this chapter may also be extended to networks having a squashing function (such as a sigmoid) after the summation.

5.1.1 Utilizing Linear Superposition

The most crucial observation to make is that the network output is the linear superposition of the individual activation functions. Thus, important properties of linear superposition can be exploited to both explain previous findings and more importantly to intelligently choose network parameters for a given learning task. Additionally, mathematical tools developed for the analysis of linear systems may be exploited. This dissertation will make use of one such tool, namely frequency domain analysis.

5.1.2 Frequency Domain Analysis

Frequency domain analysis provides a means for characterizing a signal in terms of the frequency components that it contains. Frequency domain analysis is useful for analyzing the effect of a linear system, such as a filter, on a given signal. In the frequency domain, the filter output is the product of the filter transfer function and the frequency domain representation of the input signal.

One very important fact is that in a linear system, if a frequency or range of frequencies is not present in the input signal, then it cannot appear in the output signal, regardless of the filter transfer function. Similarly, if the filter transfer function is zero at some frequency or range of frequencies, then that frequency or range of frequencies cannot appear in the output signal.

Using frequency domain analysis for neural networks is slightly different. In this case one is not filtering a signal to get a different signal; instead, one is linearly combining a collection of different functions in order to synthesize some desired function. The network input is not an input signal in a filtering sense; rather it is the independent variable from which the output signal is computed. For instance, if the input represents spatial position, then the network activation functions and output will be functions of spatial position.

The frequency domain representations of the network output, desired output, and activation functions contain a great deal of information. Here the term "frequency" is being used loosely. If the network input represents time then the frequency domain is truly frequency in radians per second. If the network input represents some other quantity, then the interpretation of "frequency" must be altered to mean radians per unit measure of input quantity.

One observation that can be made is that if the desired output function has frequency content in regions where the activation functions have no frequency content, then learning this function will be very difficult (but not impossible, since the network only provides an approximation to the desired function). One would expect that the network in this case would learn slowly, if at all. It is conjectured that the network will also be less fault tolerant.

The rationale for this conjecture is that the activation functions must have significant energy in some frequency bands (since a zero activation function would be of no use in an artificial neural network). Thus, learning not only requires that one enhance frequency components which are very small, but also cancel out frequency components which are large. This requires a very carefully chosen set of network parameters, the loss of any of which will destroy the delicate balance of the network.

5.2 Spectra of Common Basis Functions

In this section, spectra for several functions commonly used as activation functions will be derived. These spectra provide valuable insight for explaining properties observed in networks. They also allow one to choose a basis function based on the required spectral properties of the desired network output.

5.2.1 The Sigmoid Function

Members of the sigmoid family of functions are unquestionably the most commonly used network activation function for artificial neural networks. The formula for the sigmoid family of curves is given in Equation 5.1.

$$\text{Sigmoid}(t) = \frac{1}{1 + e^{-at}} \qquad (5.1)$$

The sigmoid family may be thought of as a generalized step function, however the transition from zero to one is not abrupt and the function is differentiable everywhere. The sigmoid function approaches the unit step function as the parameter 'a' approaches infinity. Additionally, the hyperbolic tangent is closely related to the sigmoid equation. The hyperbolic tangent will not be considered in this dissertation, although the derivations for the sigmoid function which follow need only be modified slightly to be applied to the hyperbolic tangent.

Performing a Fourier transform directly on the sigmoid function would be difficult. Furthermore, this approach would not strictly be valid since the sigmoid function is not a finite energy signal [Carlson, 1986]. The Fourier transform for the sigmoid function does exist in the limit however. The approach taken is to find a suitable series representation of the sigmoid function and transform this series.

If one performs simple long division on the sigmoid equation, it is readily verified that the infinite series shown in Equation 5.2 is the result.

$$\text{Sigmoid}(t) = 1 - e^{-at} + e^{-2at} - e^{-3at} \ldots \qquad (5.2)$$

Also, by long division, one can obtain the infinite series shown in Equation 5.3.

$$\text{Sigmoid}(t) = e^{at} - e^{2at} + e^{3at} \ldots \qquad (5.3)$$

Equation 5.2 is more well behaved for positive t (i.e., all terms are constant or approach zero as t becomes large), while Equation 5.3 is more well behaved for negative t. For convenience let v(t) be defined as shown in equation 5.4.

$$v(t) \equiv - e^{-at} + e^{-2at} - e^{-3at} \ldots \qquad (5.4)$$

Then the Sigmoid equation can be written as:

$$\text{Sigmoid}(t) = u(t) + v(t)\, u(t) - v(-t)\, u(-t), \qquad (5.5)$$

where u(t) is the unit step function.

Once the equation is written in this form, it is possible to exploit the following Fourier transform relationship [Carlson, 1986].

If:

$$z(t) = a_1 v(t) u(t) + a_2 v(-t) u(-t) \qquad (5.6)$$

then:

$$Z(j\omega) = (a_1+a_2) V_e(j\omega) + j(a_1-a_2) V_o(j\omega) \qquad (5.7)$$

Where: $V_e(j\omega)$ is the real part of the Fourier transform of v(t) and $V_o(j\omega)$ is the imaginary part of the Fourier transform of v(t).

In the case of the Sigmoid function, $a_1 = 1$ and $a_2 = -1$. Thus, the Fourier transform of the Sigmoid function becomes:

$$Z_{\text{Sigmoid}}(j\omega) = \boldsymbol{F}(u(t)) + j2\left(\operatorname{Im}\left(\boldsymbol{F}\left(\sum_{i=1}^{\infty}(-1)^{i}\, e^{-iat}\right)\right)\right), \qquad (5.8)$$

where $\boldsymbol{F}$ represents the Fourier transform operator and Im represents the imaginary part.

Since the Fourier transform is a linear operation, the order of the summation and Fourier transform may be interchanged. Thus, the Fourier transform of the sigmoid becomes:

$$Z_{\text{Sigmoid}}(j\omega) = \pi\delta(t) + \frac{1}{j\omega} + j2\left(\operatorname{Im}\left(\sum_{i=1}^{\infty}\frac{(-1)^{i}}{(i*a+j\omega)}\right)\right) \qquad (5.9)$$

Evaluating this function is tricky at best, and is best done numerically. One quick check that can be performed is that the equation asymptotically approaches the Fourier transform of the unit step as the parameter 'a' goes to infinity.

Figures 5.2-5.4 show plots of the spectrum represented by Equation 5.9 when the first one hundred terms of the series are evaluated[4]. The equation was evaluated for a=0.25 (Figure 5.2), a=1.0 (Figure 5.3) and a=4 (Figure 5.4). For

[4]In order to verify convergence the first fifty, one hundred and five hundred terms were evaluated. The differences in the results were far too small to be observed on the scale of the plots.

the sake of comparison the spectrum of a unit step function is also plotted. Only positive frequencies are shown, since the magnitude of the transform is an even function of frequency. Also the impulse at the origin is omitted.

It can be seen that as the parameter 'a' becomes larger, the spectrum of the sigmoid function approaches that of the step function. Also, for any finite value of 'a' the magnitude of the spectrum falls off more rapidly with increasing frequency for the sigmoid function than for the unit step function.

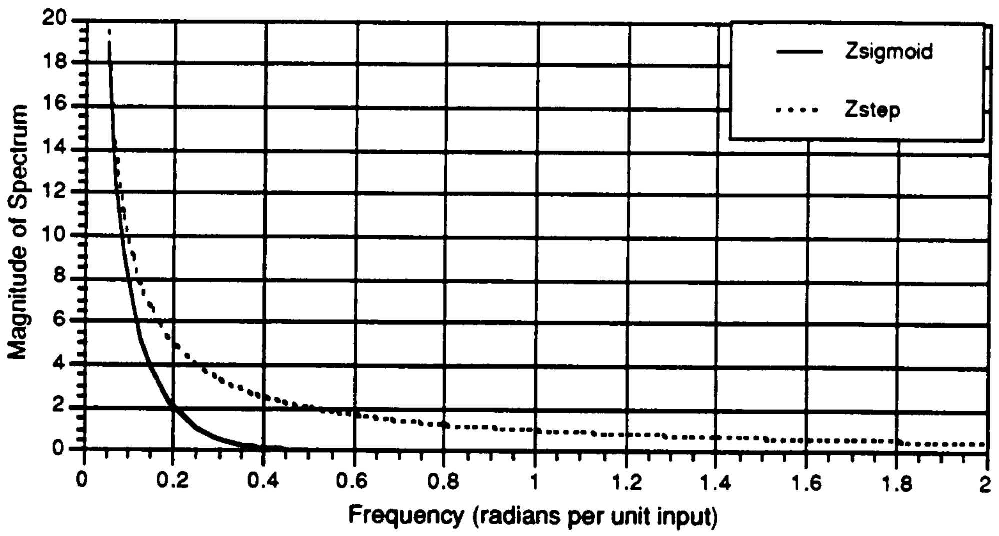

Figure 5.2 Magnitude of the Fourier transform of the sigmoid function. This data was generated by evaluating the first 100 terms of Equation 5.9 with the value 'a' set to 0.25. The impulse at the origin is not shown. The magnitude of the Fourier transform of the unit step function is plotted for comparison.

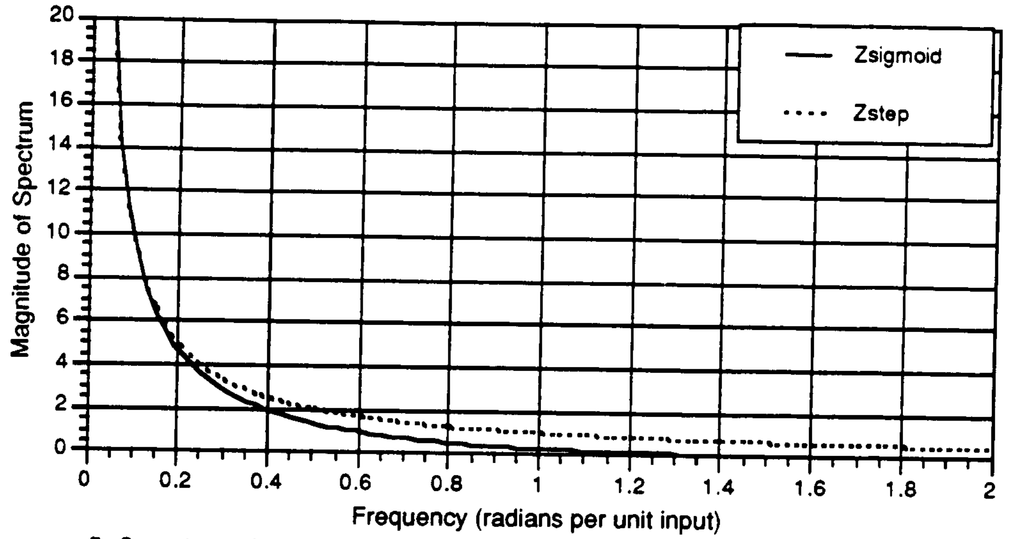

Figure 5.3 Magnitude of the frequency content of the sigmoid function. This data was generated by evaluating the first 100 terms of Equation 5.9 with the value 'a' set to 1.0. The impulse at the origin is not shown. The Fourier transform of the unit step function is plotted for comparison.

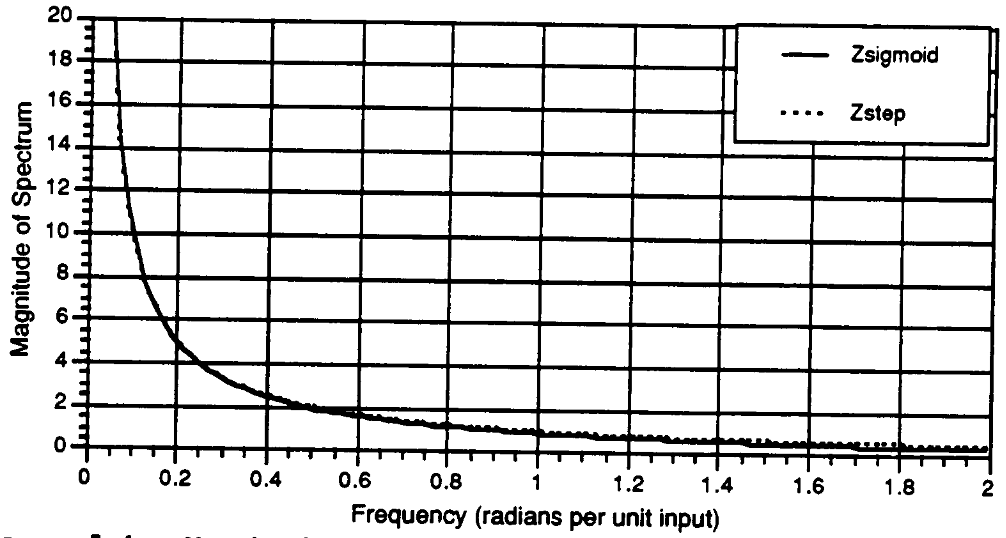

Figure 5.4 Magnitude of the frequency content of the sigmoid function. This data was generated by evaluating the first 100 terms of Equation 5.9 with the value 'a' set to 4.0. The impulse at the origin is not shown. The Fourier transform of the unit step function is plotted for comparison.

Fortunately, other commonly used activation functions lend themselves to the Fourier transform more easily than does the sigmoid function.

5.2.2 Gaussian Curves

The Fourier transform of the Gaussian function is commonly included in tables of Fourier transforms and thus will be presented here without derivation. Given a Gaussian function of the form

$$g(t) = e^{-\frac{t^2}{w^2}}, \tag{5.10}$$

its Fourier transform is given by:

$$G(j\omega) = w\sqrt{\pi}\, e^{-\frac{\omega^2 w^2}{2}} \tag{5.11}$$

Thus, the spectrum for a Gaussian function of time is a Gaussian function of frequency. A Gaussian of width w in the time domain transforms to a Gaussian (scaled by a factor of $w\sqrt{\pi}$) of width $\frac{\sqrt{2}}{w}$ in the frequency domain. Gaussian spectra for three different values of w are shown in Figure 5.5.

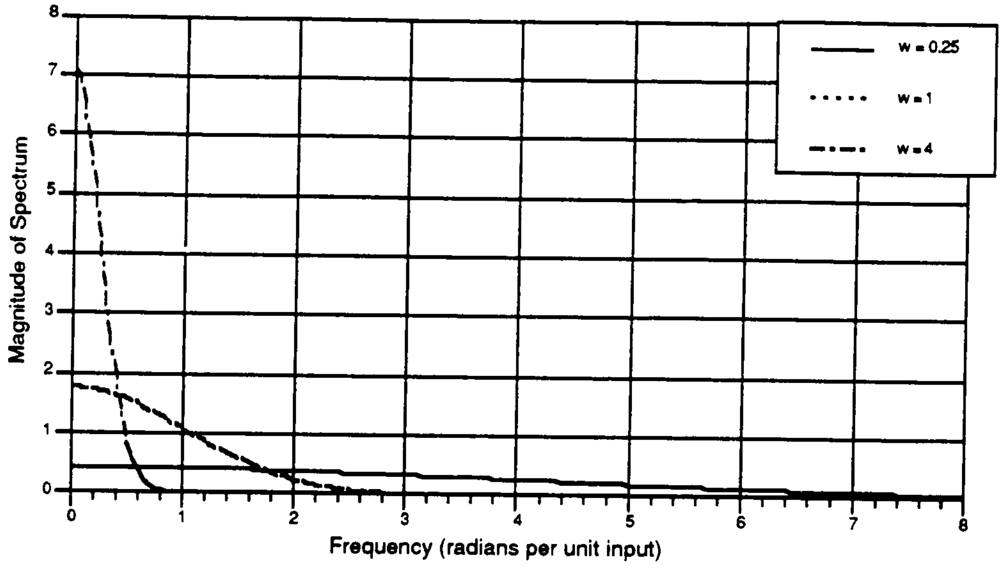

Figure 5.5 Magnitude of the Fourier transform of the Gaussian function for w = 0.25, w = 1, and w = 4.

5.2.3 The Rectangular Pulse Function

Rectangular pulses are commonly used basis functions in the CMAC neural network. The rectangular pulse function $\Pi\left(\frac{t}{\tau}\right)$ is defined as shown in Figure 5.6. It has height A and width τ.

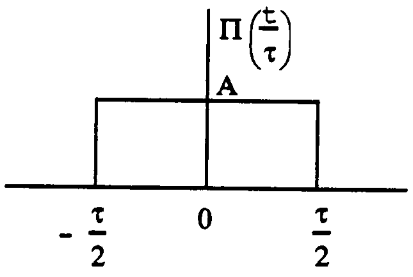

Figure 5.6 A plot of the rectangular pulse function $\Pi\left(\frac{t}{\tau}\right)$.

The Fourier transform of this function is given by:

$$F\left(\Pi\left(\frac{t}{\tau}\right)\right) = \frac{2A}{\omega}\sin\left(\frac{\omega\tau}{2}\right) \qquad (5.12)$$

A plot of the magnitude of this transform is shown in Figure 5.7.

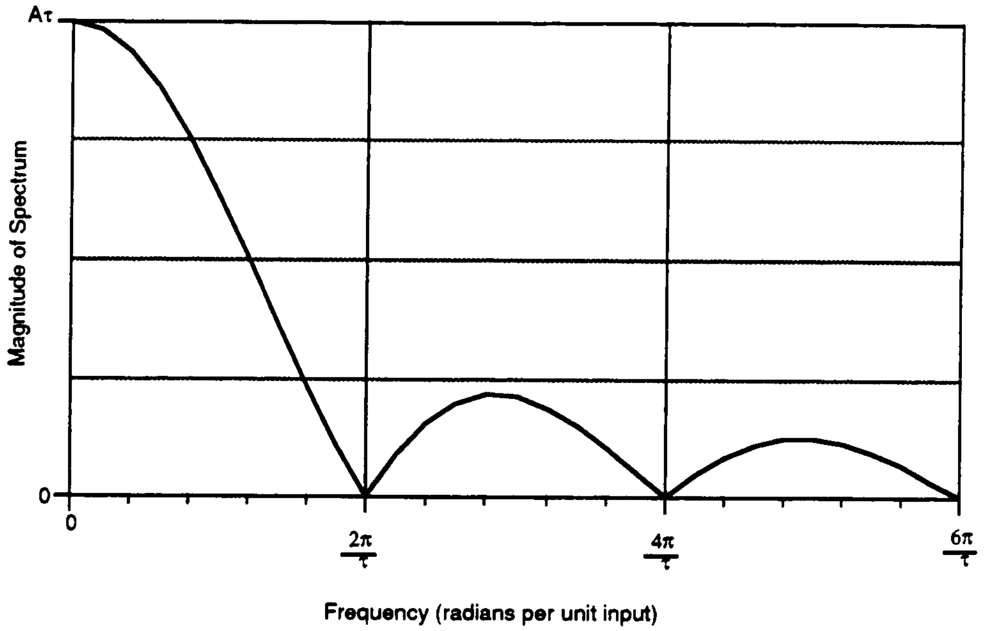

Figure 5.7 Magnitude of the Fourier transform of the rectangular pulse function shown in Figure 5.6.

5.2.4 The Triangular Pulse Function

Triangular pulses are also used (though not as commonly as square pulses) as basis functions for the CMAC artificial neural network. The triangular pulse function $\Lambda\left(\frac{t}{\tau}\right)$ is defined as shown in Figure 5.8. Notice that this function has height B and width 2τ.

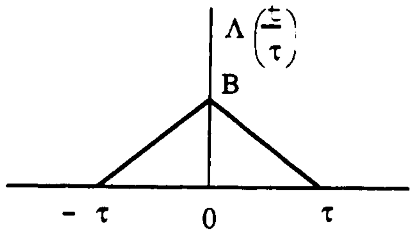

Figure 5.8 A plot of the triangular pulse function $\Lambda\left(\frac{t}{\tau}\right)$.

The Fourier transform of this function is given by:

$$\boldsymbol{F}\left(\Lambda\left(\frac{t}{\tau}\right)\right) = \frac{4\ B}{\omega^2\ \tau}\sin^2\left(\frac{\omega\ \tau}{2}\right) \qquad (5.13)$$

A plot of the magnitude of this transform is shown in Figure 5.9.

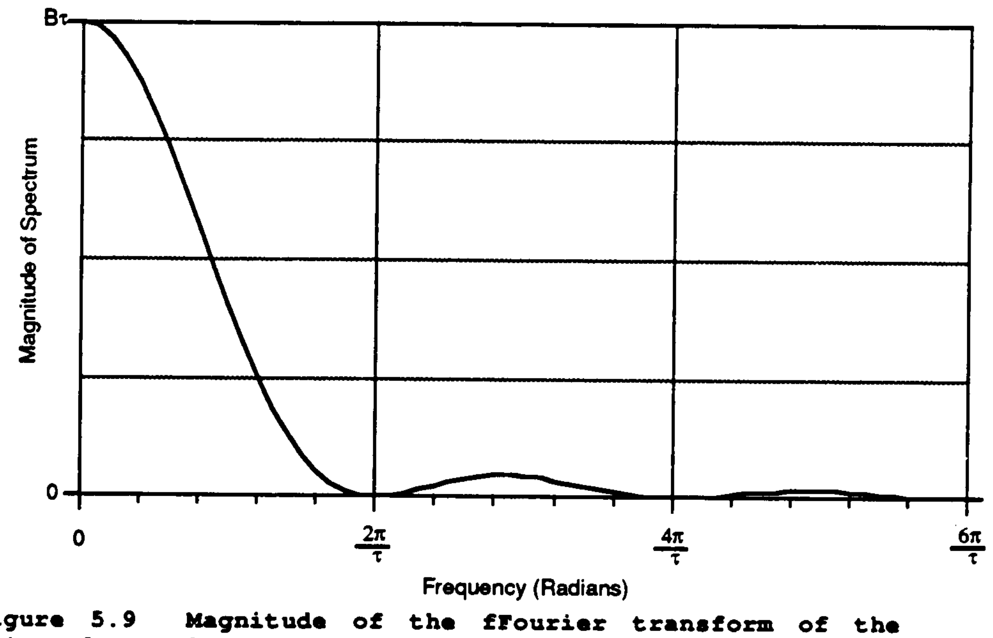

Figure 5.9 Magnitude of the fFourier transform of the triangular pulse function shown in Figure 5.8.

As can be seen, the frequency content of the triangular pulse is very similar to the frequency content of the rectangular pulse. Indeed, the zeros of these functions are identical. The magnitude of the spectrum for the triangular pulse drops off more rapidly than that of the rectangular pulse. This may be readily explained by noting the relationship between the rectangular pulse and the triangular pulse. Namely, if one convolves a rectangular pulse of height A and width τ with itself, one would get a triangular pulse of height $A^2\tau$ and width 2τ. Thus, if one lets $B = A^2\tau$, then the triangular pulse shown in Figure 5.8 is simply the convolution of two rectangular pulses as shown in Figure 5.6. A nice property of convolution is that the Fourier transform of the convolution of two signals is the product of the Fourier transforms of the two signals. In this case, as may be readily verified, the signal represented in Equation 5.13 and plotted in Figure 5.9 is the square of the signal represented in Equation 5.12 and plotted in Figure 5.7, provided that $B = A^2\tau$.

5.3 CMAC Slow Learning

During experiments with the CMAC neural network it was found [Carter, 1990A, An, 1991] that when learning sinusoidal functions, certain frequencies were very difficult to learn. These critical frequencies corresponded to the situation where the width of the rectangular receptive field (the region in which a particular weight is active) was equal to an integer multiple of the sinusoid's wavelength. Furthermore, it was found that if triangular basis functions were used, then the first critical frequency was approximately twice as high as that for rectangular basis functions with the same width and the learning speed at this critical frequency was markedly slower than at the first critical frequency when rectangular windows were employed.

All of these findings make perfect sense in the context of frequency analysis. For instance, the critical frequencies occur at frequencies where there are zeros in the basis function spectrum. Furthermore, for a given width, the triangular basis function has twice the bandwidth (between first spectral nulls) of the rectangular window; however, the frequency content of the triangular window drops off more rapidly above the first spectral null.

It is no surprise that the network has difficulty learning at the critical frequencies. It is almost more surprising that the network can learn at all at these frequencies. However, the neural network has several things going for it. The most important fact is that the network need only produce an output that is *close* to the desired output. Furthermore, this approximation need only hold for some finite interval in the input space where the network has been trained, and network behavior outside this interval is left unspecified.

Thus, CMAC slow learning is caused by attempting to learn frequencies which are absent in the network basis function. The fact that learning can occur at all can be explained by the fact that the function to be learned is a finite length segment of a sine wave, and that the network need only produce an approximation to this function. Thus, learning is much more difficult than at lower frequencies, but not impossible.

The simplest "solution" to the CMAC slow learning problem is simply to choose the receptive field width to be narrow enough that the frequencies of interest are below the first critical frequency[5]. In this way, learning can easily converge to the desired output while naturally rejecting any high frequency noise that may be present in the training data.

5.4 Experiment Results

In order to verify the usefulness of spectral analysis in predicting the properties of a neural network, an experiment was performed. In this experiment, a Gaussian radial basis function network with one hundred evenly spaced centers and each unit with a width parameter of 4.0 was trained for the same number of epoches on the same training set as the network described in Chapter 4. Every aspect of

[5] Of course, as the receptive field width becomes narrower, the required number of training exemplars for good generalization becomes larger.

the network and the training was identical to that of the network associated with Figures 4.3, 4.5, and 4.7.

As may be seen in Figure 5.5, a Gaussian function with a width parameter of 4 has very little frequency content at a frequency of one radian per unit interval. Thus, one would expect that the network would have a difficult time learning a good approximation to the sinusoid, and that the fault tolerance of the solution would be much less than that of a network with a better match between the frequency content of the basis functions and the frequency content of the function to be learned.

The results of the experiment are shown in Figures 5.10 through 5.12. Figure 5.10 shows the RMS approximation error as a function of training epoch number. As can be seen by comparing this figure to Figure 4.3, learning progresses much more slowly. The RMS approximation error ultimately reaches quite a low value after approximately 30,000 training epoches or so.

Despite the fact that the network eventually finds a reasonably good approximation, Figures 5.11 and 5.12 show that the network is much more sensitive to the loss of weights than when units with a width parameter of unity were used. Figure 5.11 shows the effect of the loss of each single network weight. The shape of the graph is similar to the graph shown in Figure 4.5, but the scales of the ordinate and abscissa are larger by approximately an order of magnitude!

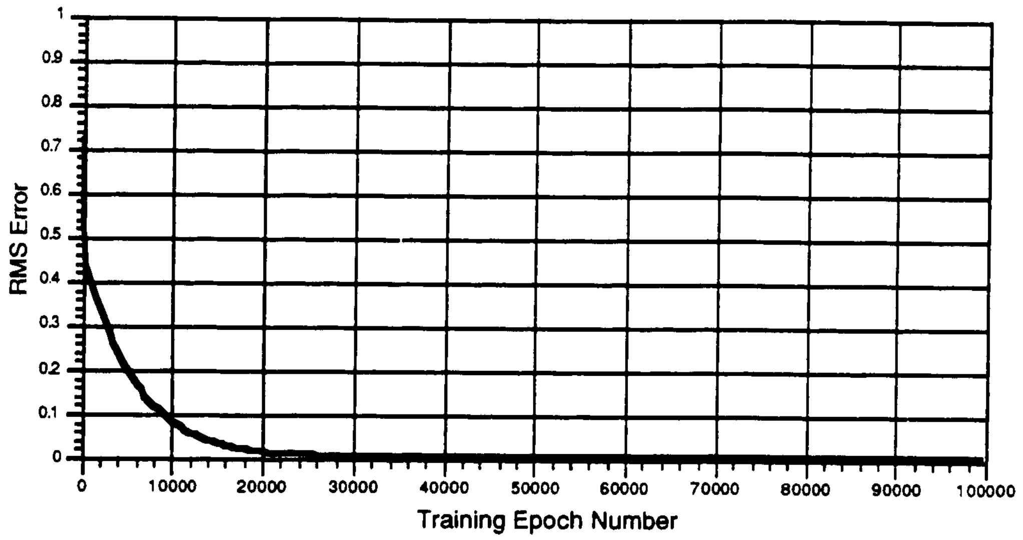

Figure 5.10 RMS error versus training epoch for a Gaussian radial basis function network with one hundred units uniformly spaced in the input domain each with a width parameter of four.

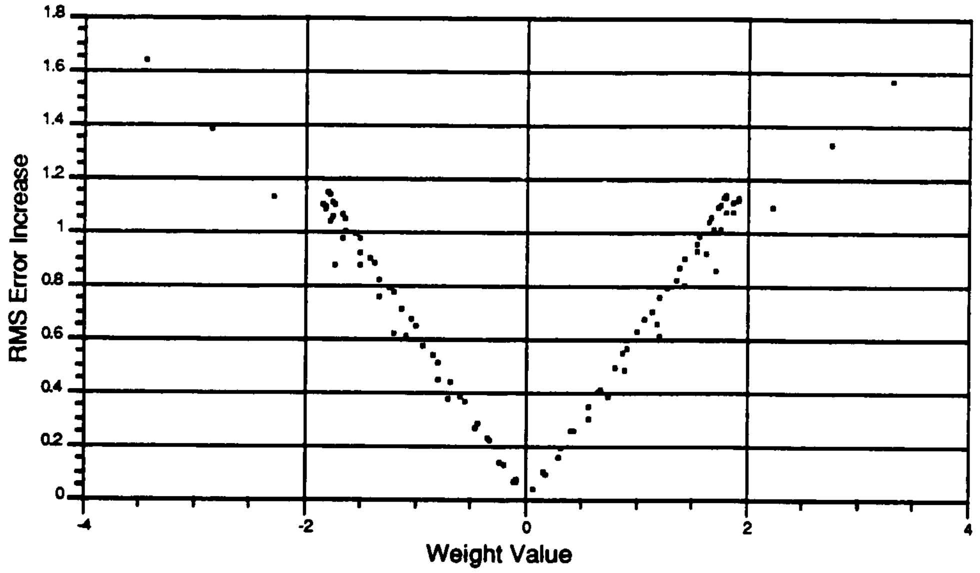

Figure 5.11 Effect on RMS error of the loss of each single weight in the Gaussian radial basis function network plotted as a function of the value of the weight lost. The network had one hundred processing elements uniformly spaced in the input domain; each one had a width parameter of four.

Furthermore, as can be seen in Figure 5.12, the shape of the AQ curve is very different from the curve shown in Figure 4.7. The loss of a single weight is sufficient to essentially ruin the approximation. The loss of additional weights in the worst case path causes even further degradation.

From these data, it is clear that a change in the width parameter is sufficient to radically alter the network learning speed and fault tolerance. Indeed, the Gaussian radial basis function network with a width parameter of four behaved more like the multilayer perceptron trained using momentum (Figures 3.2, 3.5, and 3.8) than like the Gaussian radial basis function network with a unity width parameter.

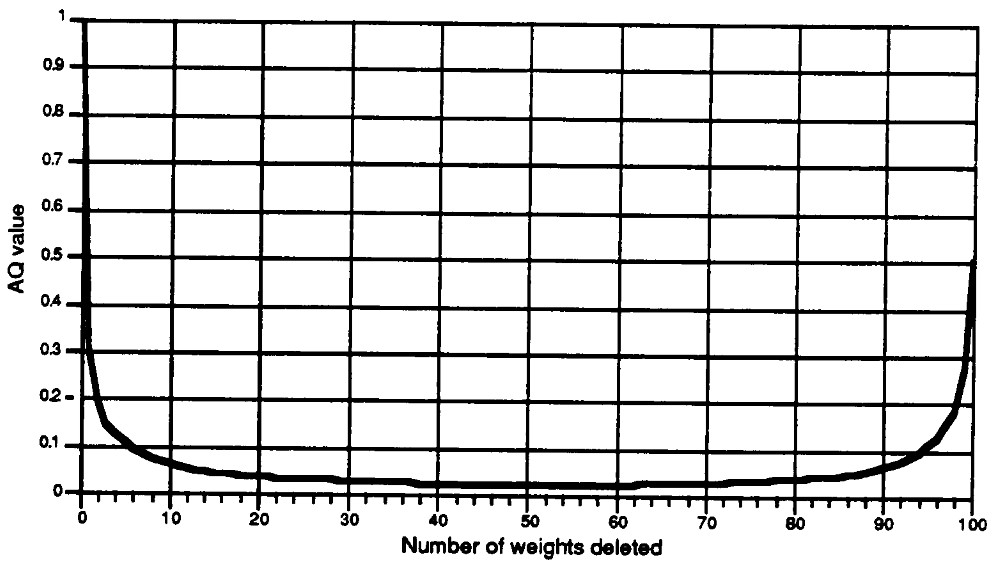

Figure 5.12 Effect on the AQ value of the Gaussian radial basis function network as weights in the worst case path are removed. The network had one hundred units uniformly spaced in the input domain; each one had a width parameter of four.

The relationship between the spectra of the activation function and the function to be learned has a direct impact on the training time and also on the fault tolerance of the trained network. The fault tolerance aspect of this relationship is discussed in the next section.

5.5 Spectral Fitting and Fault Tolerance

Many neural networks perform function approximation using the linear superposition of activation functions or compositions of activation functions. These activation functions will always favor some frequencies over others (unless delta functions are used as activation functions, and thereby reducing the neural network to a lookup table). Essentially, the process of learning requires two simultaneous constraints involving the network weights be satisfied. The first is that a set of weights must be found which accentuates frequencies which are found in the training set. Simultaneously, a set of weights must be found which causes cancellation of frequencies which are present in the activation function, but are not found in the training set. Obviously, the single set of network weights must satisfy both of these goals in order to attain a good function approximation.

When there is a strong spectral match between the activation function and the function to be learned, both tasks become easier. Smaller weights are also necessary, since frequencies present in the training set are well represented in the activation function, and frequencies to be cancelled are relatively poorly represented in the activation function. While these characteristics cannot guarantee fault tolerance, they do not actively hinder its achievement as in the case of spectral mismatch.

When there is a strong spectral mismatch between the activation function and the training set, then both learning tasks become much more difficult. Since the frequency components which must be accentuated are very small, large weights are necessary. However, these large weights must be very carefully chosen to cancel out the resulting large, undesirable, frequency components. Such a network cannot be fault tolerant, since the loss of a weight (particularly a large weight) can destroy this delicate balance.

Thus, one can conclude that although a good spectral match between the activation function and the function to be learned is not sufficient to guarantee fault tolerance, a spectral mismatch is sufficient to guarantee that a network will not be fault tolerant unless special steps are taken to control the impact of faults on the approximation. Several examples of possible methods to control the impact of faults are discussed in the next chapter.

CHAPTER VI

METHODS FOR IMPROVING FAULT TOLERANCE

In this chapter a number of possible ways that the fault tolerance of a network may be improved are discussed. The specific methods discussed are spectral methods, redundancy, good initial conditions, training with intermittent faults, and limiting the possible effect of a fault.

6.1 Spectral Methods

The results discussed in the previous chapter suggest that if one chooses an activation function that is "spectrally similar" to the function that the network is to learn, then not only will the network learn the function more rapidly, but the resulting network will also be more fault tolerant.

6.2 Engineered Redundancy

Some sort of redundancy is necessary to achieve fault tolerance. If every piece of a network is essential for operation, then the network cannot be fault tolerant, since the loss of any piece will disrupt operation. Redundancy may occur naturally in a network or it may be engineered into a network. This section deals with engineered redundancy. Promoting naturally occurring redundancy will be discussed in several subsequent sections.

Engineered redundancy has the advantage that it can be applied to existing trained networks without disrupting the network approximation properties.

6.2.1 Weight Redundancy

Weight redundancy may also be thought of as connection redundancy. With this method of redundancy, the function of an individual weight is distributed over some number of physical weights and connections. In this way, the loss of a physical weight corresponds to the partial loss of a "real" weight rather than a total loss.

One way to apply weight redundancy would be to analyze a neural network and determine the most critical weights. These weights may then be split into some larger number of weights, each only a fraction as critical as their parent weights. This method could be applied iteratively until the impact of the loss of any weight in the network was made arbitrarily small. [Izui, 1990, Carrara, 1992]

6.2.2 Node Redundancy

In a similar vein, one can split the function of a node across several physical nodes. Node redundancy is slightly more general than weight redundancy, since to duplicate a node it is necessary to duplicate (and possibly scale) any weights associated with that node.

Like weight redundancy, node redundancy can be applied to an existing network with no change in its fault-free performance. Nodes that produce the greatest loss of performance can be split into multiple physical nodes (with their weights duplicated or scaled accordingly), each of which has less impact when lost.[Phatak, 1990]

6.2.3 Network Redundancy

Finally, the most general method of redundancy is to simply duplicate the entire network some number of times and average the resulting network outputs together.

Obviously, network redundancy is the most general type of redundancy since it involves duplicating weights and nodes. It is also the most wasteful of resources, since even unimportant portions of the network are duplicated. It is also the simplest to implement since no analysis is necessary. One merely needs to train a network to a desired degree of accuracy and then duplicate this network as many times as possible (or practical). This philosophy is analogous to the N-fold redundancy schemes widely used in conventional fault tolerant computation.

6.2.4 Criticism of Engineered Redundancy

There are two primary criticisms of engineered redundancy in neural networks. The first is that it does not mimic biological systems. The second is that if one adds more components, one would expect more faults to occur assuming a constant failure probability. Thus even if the effect of each fault is reduced, the overall reliability is unchanged.

The criticism that engineered redundancy does not mimic biological neural systems may or may not be valid. The fact is that we have a very poor understanding of what biological systems do and what they don't do, especially at the microscopic level in which any explicit redundancy would be observable. Furthermore, merely because artificial neural networks were inspired by biological systems is no reason that they should be limited by the fault tolerance mechanisms of biological systems. Researchers should feel free to borrow good solutions from nature without constraining themselves unnecessarily. Thus, engineered redundancy may or may not be used by nature, but even if it isn't used by nature it should still be considered a valid tool for use in artificial neural networks.

The second criticism, that adding more components would merely cause more faults to occur thus cancelling the benefit of less degradation per fault, is valid only when all faults cause the same degradation in performance. As long as some parameters are more critical than others, redundancy can not only reduce the amount of degradation caused at each step in the worst case path, but also makes the worst case path much less likely to be followed since the number of possible paths grows combinatorially with the number of network parameters.

6.3 Good Initial Conditions

Since the Gaussian radial basis function network was initialized in a "well chosen" manner (i.e., evenly spaced centers, unit width parameters, and zero weights), and the multilayer perceptron was initialized randomly, it was hypothesized that the difference in initial conditions was partly responsible for the differences in fault tolerance of these networks. Experiments were conducted to test this hypothesis.

An initialization method for the multilayer perceptron was chosen to be similar to that used for the Gaussian radial basis function network. The bias term for a processing element of the multilayer perceptron controls the location of the point of the symmetry of the sigmoid activation function for that unit and thus is analogous to the center parameter of the Gaussian function. Thus, the bias terms were chosen to be uniformly distributed across the region of interest in the input space. The weights between the input and the hidden layer of units in the multilayer perceptron have the effect of spreading or sharpening the sigmoid function (i.e., very large weights make the sigmoid similar to a hard limiter, while very small weights make the transition very gradual). Thus, the weights in the first layer are analogous to the width parameter of the Gaussian radial basis function. These weights were cnosen to alternate between +1 and -1 to provide a reasonable width value and to cause half of the sigmoids to open to the right and half to open to the left. Finally, each weight between the hidden layer and the output was initialized to be the value of the sine of the associated bias parameter.

Following initialization, the network was trained using ordinary backpropagation. This procedure did indeed improve the fault tolerance slightly, but to nowhere near the level of the Gaussian radial basis function network. Furthermore, if was found that longer training times lessened the difference in fault tolerance between the randomly initialized network and the network initialized using "well chosen" values. This implies that backpropagation not only fails to produce fault tolerant solutions, but also that increased training with backpropagation tends to diminish any existing fault tolerance.

Results for the multilayer perceptron trained with "well chosen" initial conditions are shown in Figures 6.1 and 6.2.

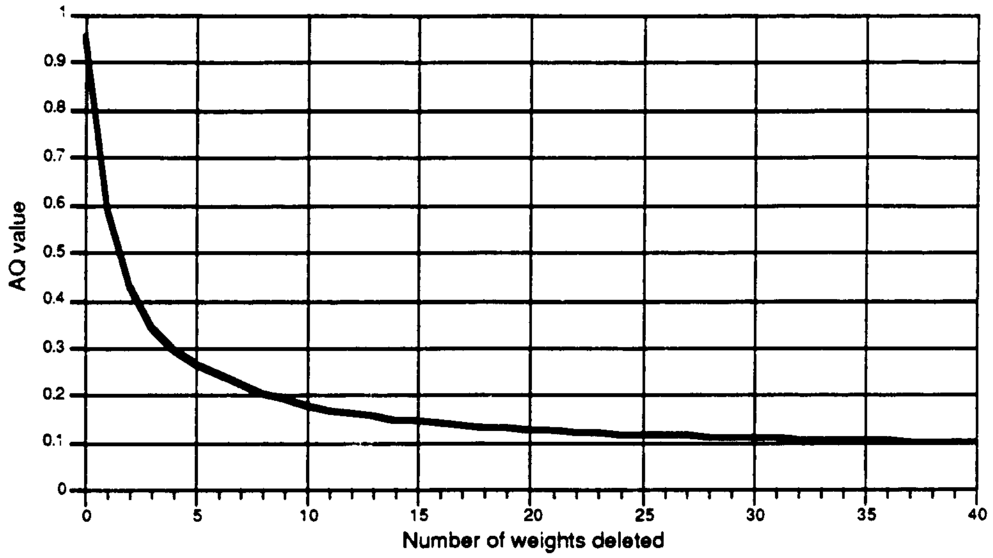

Figure 6.1 AQ curve for the multilayer perceptron initialized with well chosen weights and trained for 10,000 epoches.

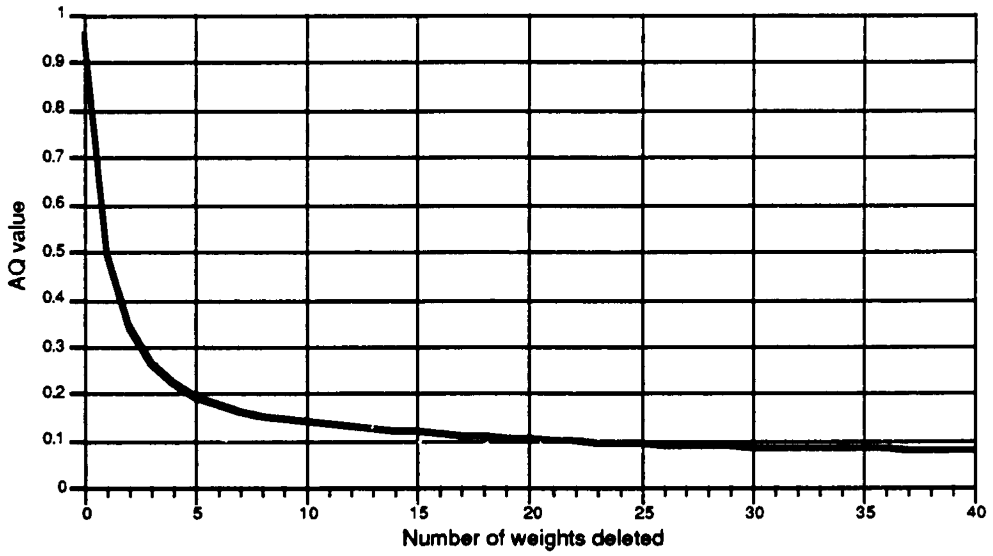

Figure 6.2: AQ curve for the multilayer perceptron initialized with well chosen weights and trained for 100,000 epoches.

Only the first twenty percent of the worst case path is shown in Figures 6.1 and 6.2 so that the differences in the curves may be more clearly seen. The fault tolerance is improved over the randomly initialized network, although performance is still less than one would hope.

6.4 Training With Intermittent Faults

This method of improving fault tolerance was proposed by [Séquin, 1990]. During training, some (fixed) number of nodes are randomly chosen to be disabled at each training point. The network is then trained without these nodes present, and then the nodes are re-enabled. At the next training point a new set of randomly chosen nodes are disabled and the process is repeated. Training with intermittent faults helps prevent highly critical units from forming, since when such a unit is disabled, the error would be large and the weights in the remainder of the network would be adjusted to compensate for the loss. This method has been found to improve the network fault tolerance at the expense of degraded fault-free performance.

6.4.1 Multilayer Perceptron Results

Figures 6.3 through 6.6 show the effect on the worst case path fault tolerance of training the multilayer perceptron with intermittently failing nodes. The number of randomly chosen nodes was varied between one and twenty.

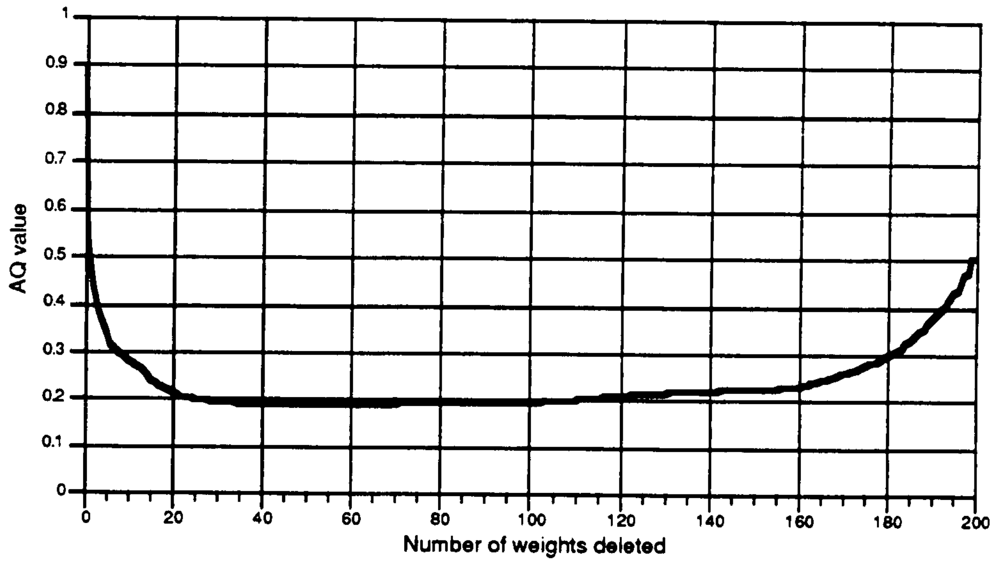

Figure 6.3: Effect on the AQ curve of a multilayer perceptron when one intermittently failing unit is present during training.

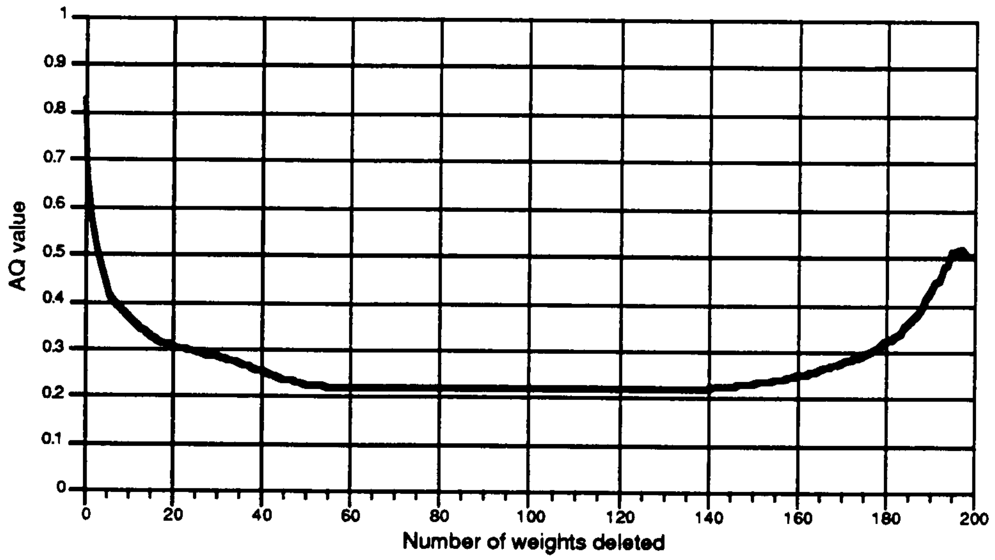

Figure 6.4: Effect on the AQ curve of a multilayer perceptron when five intermittently failing units are present during training.

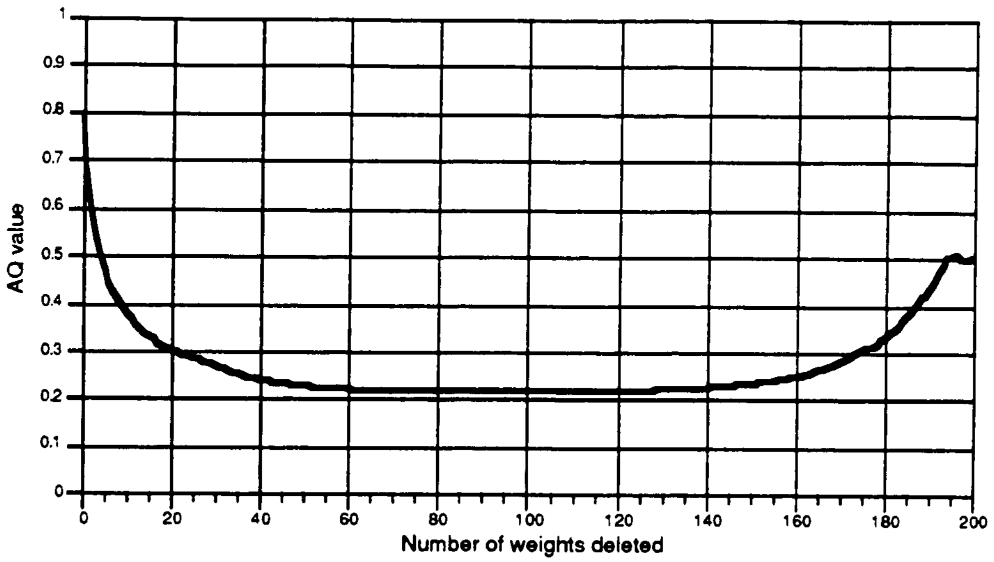

Figure 6.5: **Effect on the AQ curve of a multilayer perceptron when ten intermittently failing units are present during training.**

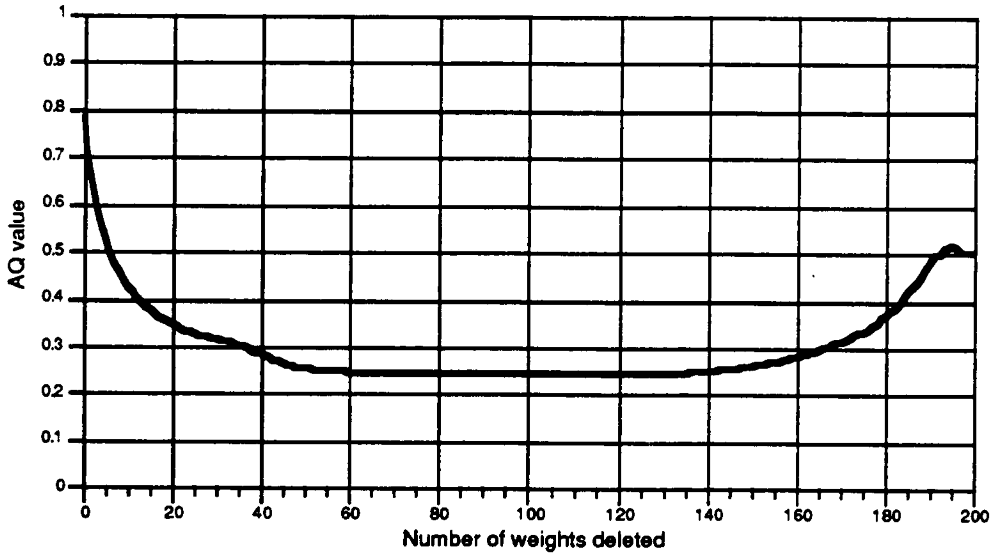

Figure 6.6: **Effect on the AQ curve of a multilayer perceptron when twenty intermittently failing units are present during training.**

Even when the number of intermittent failures was limited to one randomly chosen unit per epoch, the accuracy to which the multilayer perceptron could learn the function was significantly degraded. Nevertheless, the fault tolerance was improved. The network could tolerate the loss of several weights before the AQ value dropped below 0.5. Furthermore, the lowest value of approximation quality observed on the worst case path increased as more and more intermittent faults were introduced during training.

6.4.2 Gaussian Radial Basis Function Results

When the method of training with intermittent faults was applied to the Gaussian radial basis function network, the results were much more impressive. The general trend was again toward improved fault tolerance at the loss of fault-free approximation quality. However, the relative behavior of the two factors differs from that observed with the multilayer perceptron. The results for the Gaussian radial basis function network are shown below in Figures 6.7 through 6.10.

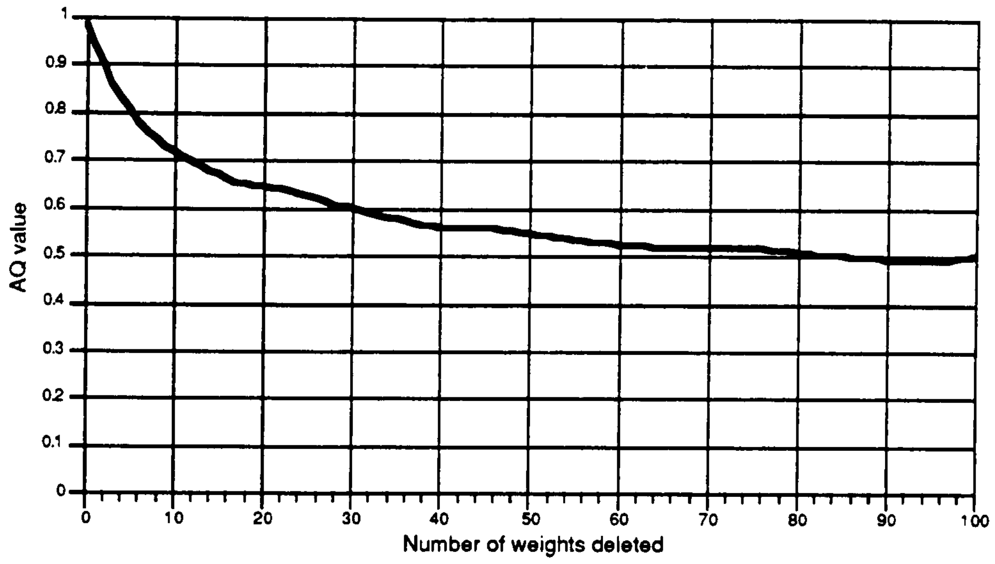

Figure 6.7: Effect on the AQ curve of a Gaussian radial basis function network when one intermittently failing unit is present during training.

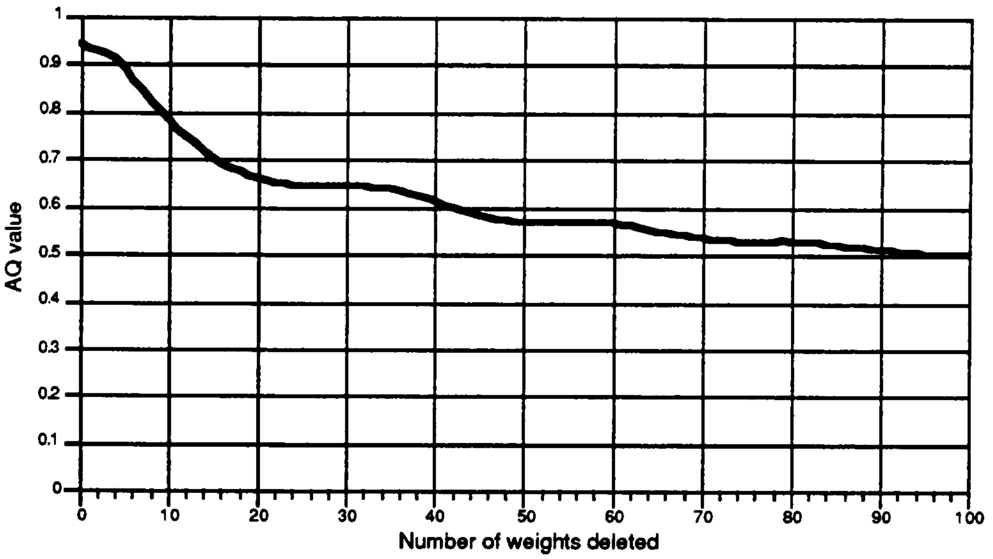

Figure 6.8 Effect on the AQ curve of a Gaussian radial basis function network when five intermittently failing units are present during training.

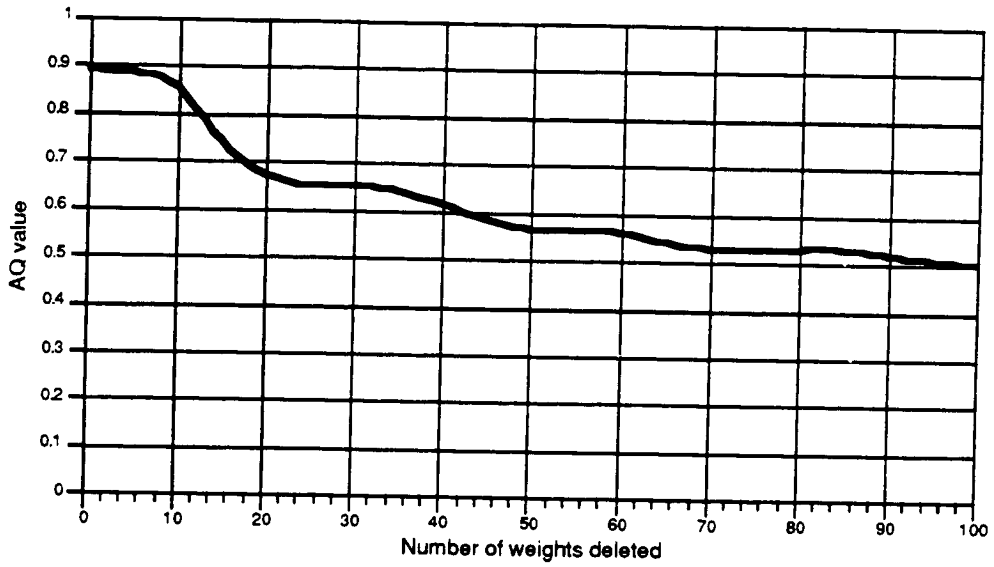

Figure 6.9 Effect on the AQ curve of a Gaussian radial basis function network when ten intermittently failing units are present during training.

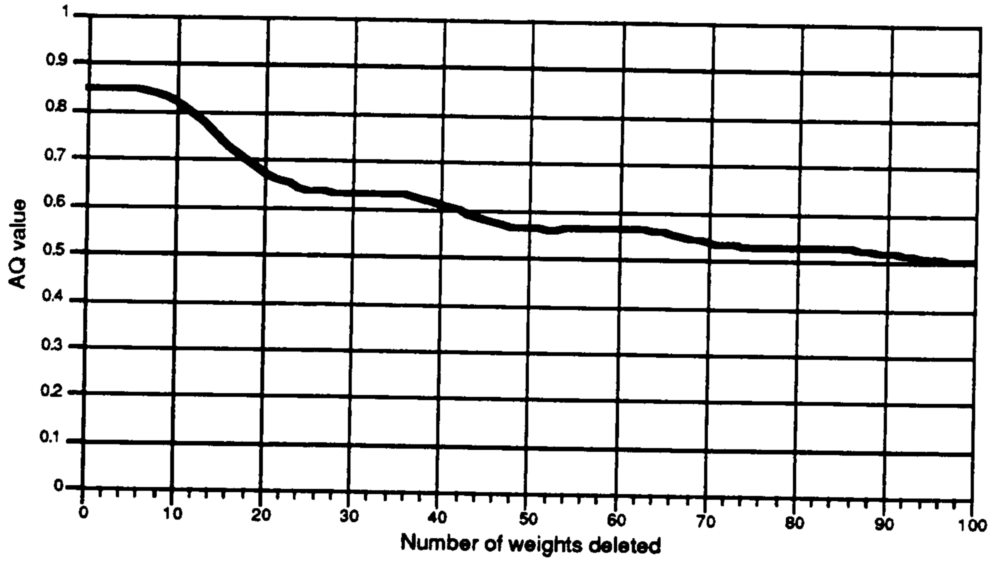

Figure 6.10: Effect on the AQ curve of a Gaussian radial basis function network when twenty intermittently failing units are present during training.

The presence of a single intermittent failure during training (chosen randomly at each presentation of a training pattern) had very little impact on the fault-free performance of the Gaussian radial basis function network. Furthermore, the AQ curve dropped off somewhat more slowly as the worst case weight loss path is followed.

Increasing the number of intermittently failing units caused the fault-free approximation quality to diminish, although this effect was much less than was observed with the multilayer perceptron. As more and more intermittent failures were introduced, something remarkable happened. The AQ curve became essentially flat for some number of weight deletions (nearly 10% of the network weights). Thus, for the loss of a small number of weights, *weights which caused the maximum degradation causes essentially no degradation* with respect to fault free performance.

6.4.2.1 Single Weight Fault Tolerance

The remarkable fault tolerance results from this training method can be readily explained. If one observes the graphs (Figures 6.11 through 6.14) showing the effect on RMS error caused by the loss of each network weight individually, one will note that there is a distinct qualitative change from those presented in Chapter 4.

The network represented in Figure 6.11 is identical to the network represented in Figure 4.5, except for the presence of a single intermittently failing unit during training. As can be seen by comparing Figure 6.11 to Figure 4.5, training with a single intermittently failing unit has two effects on the network. The shapes of the two graphs are quite similar, but the magnitudes of the largest weights in the intermittently failing node case are less than one third of those in the network trained by conventional gradient descent, and the maximum RMS error increase is approximately one fourth as large.

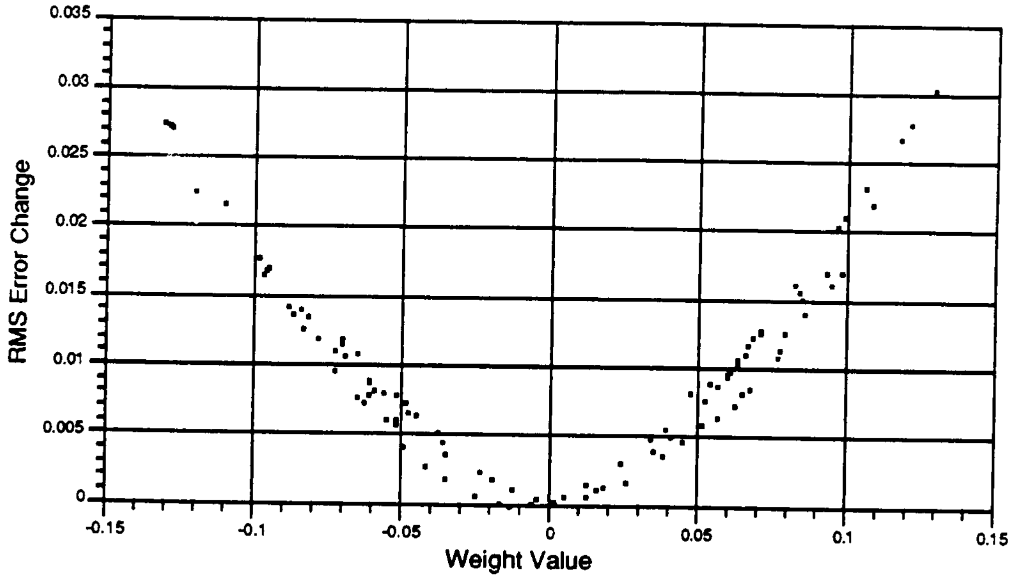

Figure 6.11 Effect on RMS approximation error of the loss of each single weight in a Gaussian radial basis function network trained with one intermittently failing unit. The network had one hundred processing elements uniformly spaced in the input domain; each had a unity width parameter.

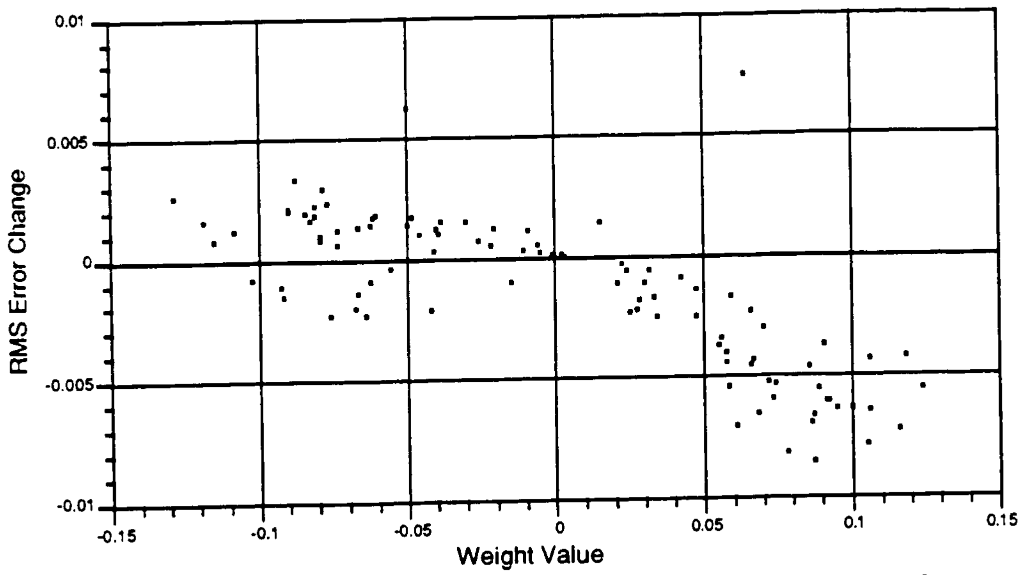

Figure 6.12 Effect on RMS approximation error of the loss of each single weight in a Gaussian radial basis function network trained with five intermittently failing units. The network had one hundred processing elements uniformly spaced in the input domain; each had a unity width parameter.

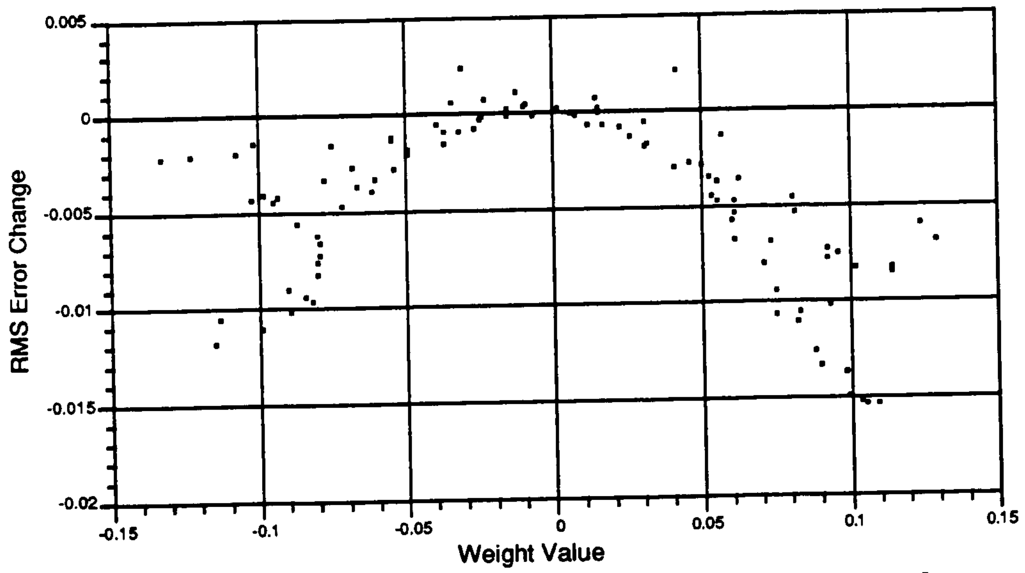

Figure 6.13 Effect on RMS approximation error of the loss of each single weight in a Gaussian radial basis function network trained with ten intermittently failing units. The network had one hundred processing elements uniformly spaced in the input domain; each had a unity width parameter.

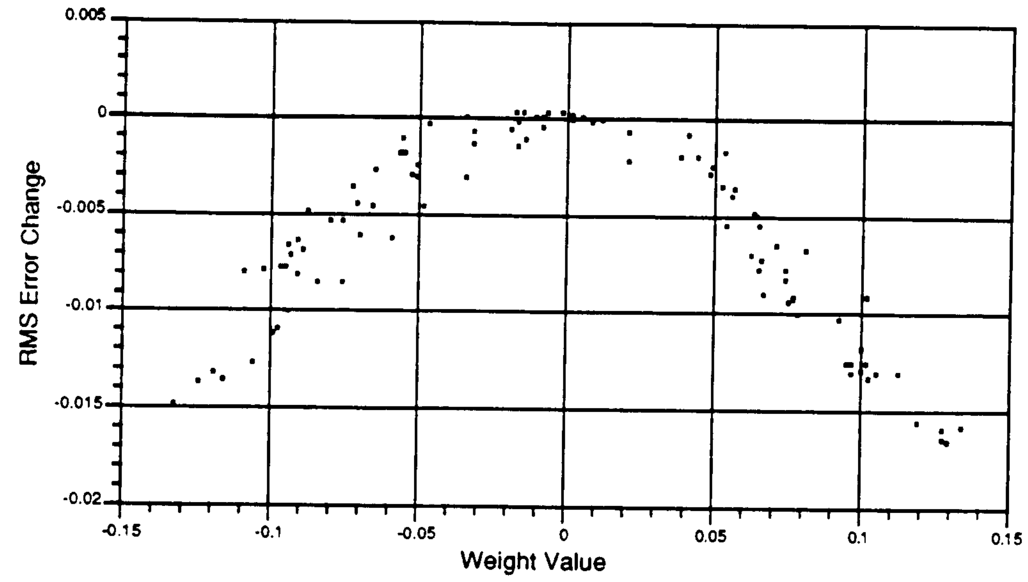

Figure 6.14 Effect on RMS error of the loss of each single weight in a Gaussian radial basis function network trained with twenty intermittently failing units. The network had one hundred processing elements uniformly spaced in the input domain; each had a unity width parameter.

As the number of intermittently failing units permitted during training was increased (Figures 6.12, 6.13, and 6.14) the behavior of the network in the presence of faults changed dramatically.

Although the range of weight magnitudes remained very similar to that of the network trained with a single intermittently failing unit, the maximum degradation caused by the loss of a single weight became much smaller (as did the absolute value of the weight which caused the maximum degradation). The second important difference in network behavior was that the "degradation" caused by the loss of larger (absolute value) weights became negative, meaning that the *loss* of these weights actually *improved* the network performance. This represents a "sensitivity inversion" wherein the weights are biased away from the minimal error fault-free solution toward a much more robust solution.

6.4.2.2 Partial Fault Tolerance

The partial fault tolerance graphs for the Gaussian radial basis function network trained with intermittently failing units also have very interesting features.

Graphs showing the effect on RMS approximation error of adding and subtracting ten percent from the value of each weight are shown in Figures 6.15 and 6.16. Figure 6.15 represents the Gaussian radial basis function network trained with a single intermittently failing unit and Figure 6.16 the network trained with twenty intermittently failing units. As can be seen, the general trend is that for a ten percent decrease in weight value the RMS error improves slightly, while for a ten percent increase in weight value the RMS error worsens slightly.

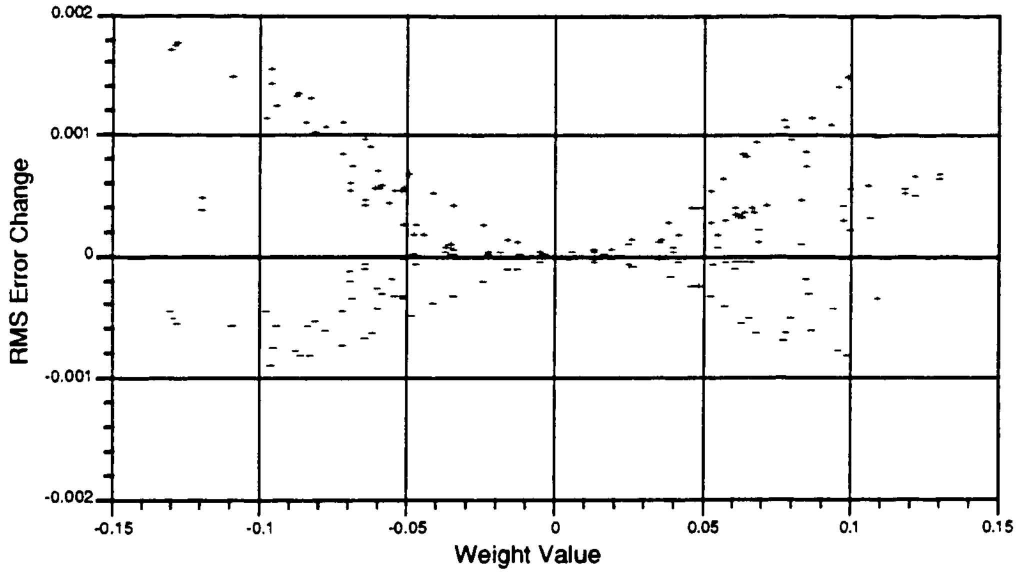

Figure 6.15 Effect on RMS approximation error of a ±10% change of each single weight (individually) in the Gaussian radial basis function network trained with one intermittently failing unit. The network had one hundred processing elements uniformly spaced in the input domain; each had a unity width parameter.

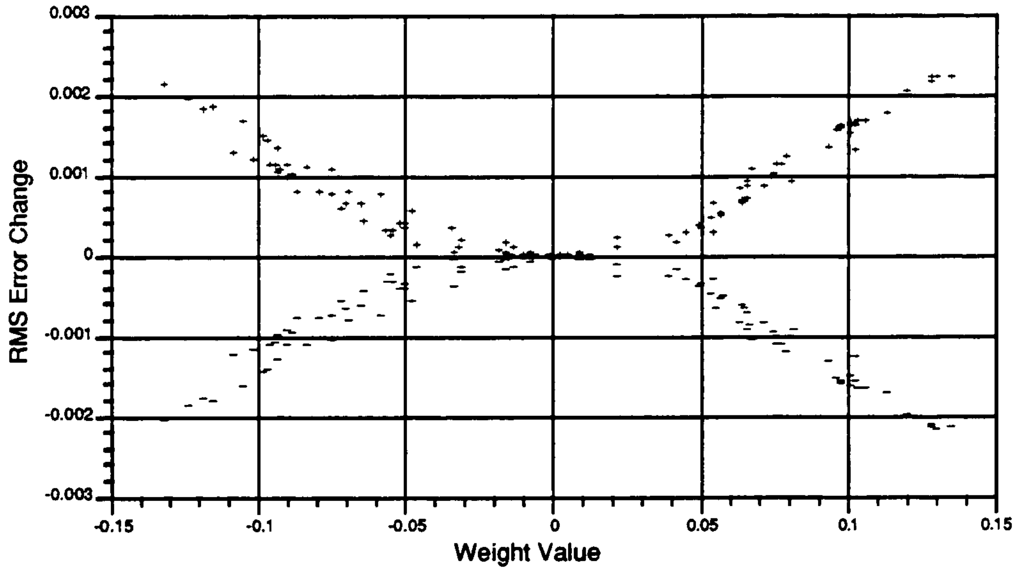

Figure 6.16 Effect on RMS error of a ±10% change of each single weight (individually) in the Gaussian radial basis function network trained with twenty intermittently failing units. The network had one hundred processing elements uniformly spaced in the input domain; each had a unity width parameter.

For the case of training with a single intermittently failing unit, the RMS error degradations were slightly more pronounced than the improvements, while for the case of training with twenty intermittently failing units the two curves have nearly "mirror-image" symmetry.

By referring to Figure 6.16, one can see that any decrease in absolute value of any weight caused network performance to improve, and any increase in absolute value of any network weight caused performance to worsen. Thus, one can conclude that training with intermittently failing nodes biases the network toward a solution where each weight is *larger* in absolute value than is necessary. Interestingly, however, the network weights of the network trained with twenty intermittently failing units were *smaller* in magnitude than those in an identical network trained without any intermittently failing units (see Figure 4.9).

<u>6.4.3 Comparing the Multilayer Perceptron and the Generalized Radial Basis Function Network</u>

The Gaussian radial basis function fault tolerance results are made even more remarkable by the fact that the multilayer perceptron did not behave similarly. The AQ curves for the multilayer perceptron trained with intermittent unit failures (Figures 6.3 though 6.6) do not display the "flattening" effect apparent in Figures 6.9 and

6.10. Thus, for the multilayer perceptron the loss of the most critical weight always produces performance degradation.

For the sake of comparison with the Gaussian radial basis function network, the single weight fault tolerance (Figure 6.17) and the partial fault tolerance (Figure 6.18) are given for the multilayer perceptron trained with twenty intermittently failing units.

Training with intermittently failing units greatly improves the single weight fault tolerance of the multilayer perceptron as can be seen by comparing Figure 6.17 with Figure 3.4. Despite similar weight magnitudes between the two networks, the network trained with twenty intermittently failing units had a maximum RMS error increase which was at least an order of magnitude less than the maximum RMS error increase of the network trained without intermittently failing units. Nevertheless, there was still a noticeable degradation in approximation accuracy if the worst case weight was lost.

On the other hand, the Gaussian radial basis function network trained with twenty intermittently failing units had a maximum RMS error increase which was virtually zero from the loss of a single weight.

The partial fault tolerance graph for the multilayer perceptron (Figure 6.18) is also not as well behaved as its Gaussian radial basis function counterpart (Figure 6.16). While Figure 6.16 has almost perfect symmetry, with ten percent changes in weight values causing equal and opposite changes in RMS error, Figure 6.18 lacks this symmetry. Furthermore, there were many more weights in the multilayer perceptron which caused increased rather than decreased error, and the increases tended to be larger in magnitude than the decreases.

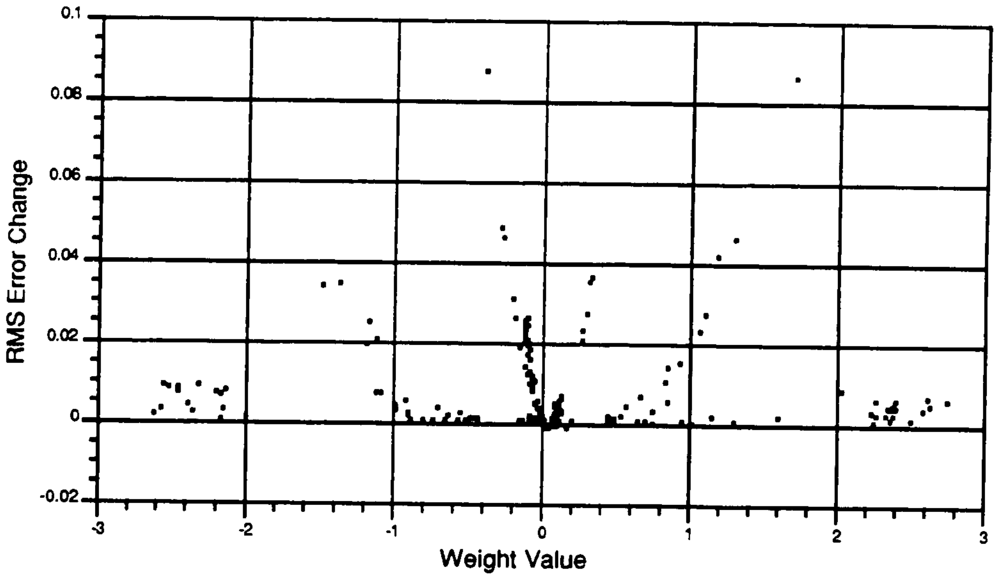

Figure 6.17 Effect on RMS approximation error of the loss of each single weight (individually) in a multilayer perceptron trained with twenty intermittently failing units. The network had one hundred processing elements.

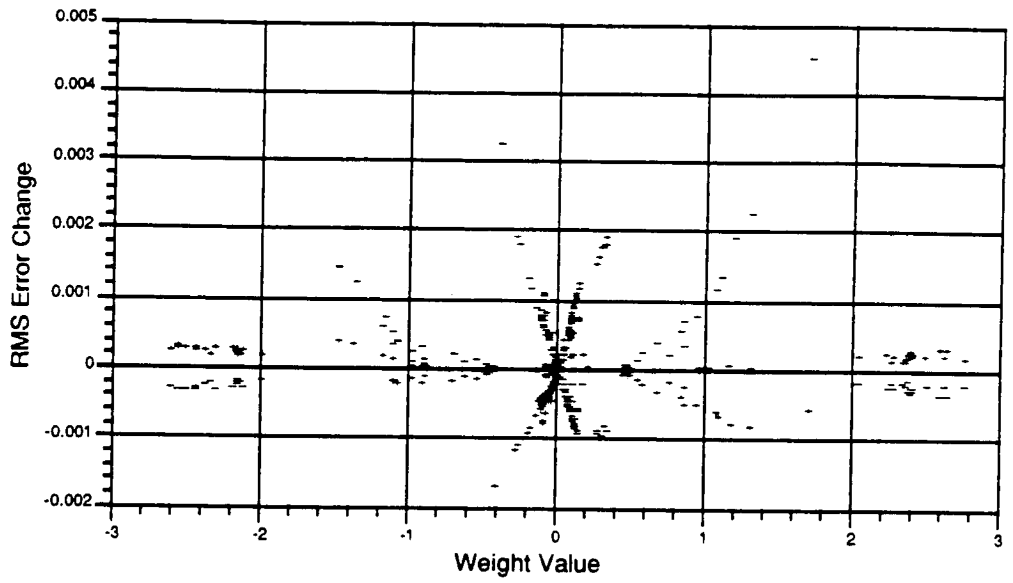

Figure 6.18 Effect on RMS approximation error of a ±10% change of each single weight (individually) in a multilayer perceptron trained with twenty intermittently failing units. The network had one hundred processing elements.

One can conclude that training with intermittent node failures significantly improves the fault tolerance of both the multilayer perceptron and the Gaussian radial basis function network. The improvement in the case of the Gaussian radial basis function network was much more pronounced than that for the multilayer perceptron, despite the fact that the Gaussian radial basis function network far outperformed the multilayer perceptron in terms of fault tolerance even when no special training methods were used.

6.5 Limiting the Possible Effect of a Fault

Another approach to assuring fault tolerance is to strive to minimize the impact of a network fault. Three methods are discussed, viz., limited dynamic range weights, continuous training, and sparse connectivity.

6.5.1 Limited Dynamic Range Weights

It is usually the case that the weights which cause large increases in approximation error when lost are large in magnitude. In fact the larger the weight value is, the larger the increase in RMS error it causes when lost (except in some networks trained with intermittently failing nodes). Thus, it is reasonable to attempt to improve network fault tolerance by simply limiting the network to small weights.

Clay and Séquin [Clay, 1991] have reported success with such a method. They developed a network of localized basis function units, each of which had an identical weight and width. The researchers drew an analogy with building a structure with bricks, each brick is the same size, but one may build many interesting structures by stacking the bricks appropriately.

6.5.2 Continuous Training

Still another philosophy is to train the network continuously. Thus, a fault may occur which seriously degrades performance; however, provided the network has sufficient computational capacity remaining, it will eventually be re-trained to compensate for the deficit.

One problem with this method is that it has been found [Holmes, 1991] that re-training a network after a fault has occurred may be a difficult problem. In fact, it has been reported that in some cases it is faster to reinitialize[6] the network and completely retrain, rather than re-train from a faulted state.

6.5.3 Sparse Connectivity

Sparse connectivity is an approach to limiting the possible effect of a fault which is applicable only to networks having multiple outputs or multiple layers of processing elements. Many, if not most, artificial neural networks in use are fully connected. A fully connected layered network is one in which the output of a node in a given layer drives the input of every node in the next layer. A fully connected non-layered network is one in which the output of a node drives the input to every other node in the network.

Sparse connectivity, on the other hand, means that a network has relatively few connections. It is unclear if sparse connectivity actually improves network fault tolerance, but the argument in favor of it goes as follows [Bolt, 1991A]: In a sparsely connected network, a fault may occur, and may even significantly impact a portion of the network. However, because of sparse connectivity, the

[6] Note that reinitialization does not make the fault go away.

majority of the network would be completely unaffected and thus the overall impact of the fault would be small, provided that the calculation is well distributed across the entire network.

CHAPTER VII

CHOOSING NETWORK PARAMETERS FOR IMPROVED FAULT TOLERANCE

It has been shown in this dissertation that an appropriate choice of network parameters and initial conditions can not only shorten learning time, but can also improve the network's fault tolerance. Thus, one has a win-win situation. In this chapter strategies for choosing network parameters to improve fault tolerance are presented.

7.1 Choosing a Suitable Activation Function

Choosing a network activation function (or basis function) is very important. Ideally, the activation function should be spectrally similar to the function to be learned. This leads to good learning performance [Pati, 1990, Sanner, 1991] and also fault tolerance. If one does not know *a priori* the function to be learned, one must make some assumptions regarding function smoothness and/or spectral content.

Generally speaking, the sigmoid function is almost always a bad choice unless one is interested only in approximating extremely low frequency functions. The magnitude of the spectrum of the sigmoid function is very badly behaved, since it tends to infinity for very low frequencies, and drops off very rapidly.

A Gaussian activation function, by contrast, is very well-behaved in the frequency domain. It has nice lowpass properties (as do most physical processes) with a bandwidth that is controlled by the width parameter.

The rectangular and triangular basis functions also have lowpass properties; however their spectra do not approach zero monotonically. These activation functions have the advantage of simplicity in implementation, especially in the CMAC architecture [Miller, 1990].

For each of the activation functions there is a trade-off between the bandwidth of the activation function (in the frequency domain) and the width of the activation function in the input domain. In general, to cover a wide band of frequencies one needs a very narrow activation function. This, in turn, means that one must space the activation functions very close together and one must have very many training points. On the other hand, if one uses wide activation functions, one can span the input space using far fewer processing elements, but the network will be well suited only for learning relatively low frequency functions and as shown in Section 5.4, the fault tolerance suffers.

Consider, for example, the rectangular activation function. If one chooses an infinitely wide pulse, one only needs a single processing element to span any input space.

Of course, the network could only learn a constant function approximation (zero frequency). If, on the other hand, one chooses rectangular basis functions of vanishingly small width, one could learn any function regardless of frequency content. Unfortunately, one would also need an infinite number of processing elements and an infinite number of training points to span even a finite space.

If one thinks about activation function width in a slightly different way, then the previous argument also applies to the sigmoid activation function. Here the "width" is the transition region between zero and one, and is determined by the weights in the input layer or a sigmoid gain parameter. If these weights are very small, the transition region is very wide and the magnitude of the basis function spectrum drops off very rapidly. If the weights are very large, then the sigmoid approaches a unit step function. The transition region becomes very narrow and the high frequency content increases (although it still does not afford good performance in the fault tolerance context).

Of course, if one is not interested in having lowpass properties in an activation function, one could choose a wavelet, or more generally a frame [Strang, 1989, Pati, 1990]. These functions have bandpass (rather than lowpass) properties.

7.2 Choosing Initial Centers/Biases

Lacking any specific knowledge about the function to be learned, it is best to spread the units evenly across the space of interest. Of course, if one has specific knowledge about the function to be learned (for instance, if the function is identically equal to zero in some region of the space) one should take this into account when choosing initial centers or biases.

7.3 Choosing Initial Weights

Generally speaking, initializing weights in the output layer of a network to zero is a good choice. Since the weights in the input layer of the multilayer perceptron have the effect of controlling the transition steepness (or width) of the sigmoid function as well as its orientation in the input space, it makes sense to initialize these weights to small random values, as is commonly done. This causes one to have a set of wide sigmoid activation functions opening at random orientations in the space[7].

Bias terms in the multilayer perceptron tend to be treated as weights. However their function is more analogous

[7] The sigmoid function effectively selects a half space whose orientation is determined by the weights in the input layer. It is desirable to initialize these weights randomly to select as many different half spaces as possible.

to the center parameter of a localized basis function network. Thus, the biases should be initialized as discussed in the previous section to spread the functions uniformly throughout the space of interest.

CHAPTER VIII

WRAP UP

8.1 Summary and Conclusions

Despite the fact that biological neural networks display remarkable fault tolerance, artificial neural networks do not automatically inherit these fault tolerance properties. The fault tolerance of artificial neural networks is an issue which has received relatively little attention. This is unfortunate not only because artificial neural networks are less fault tolerant than is widely believed, but also because there is evidence that improved fault tolerance leads to improved network learning and generalization performance as well.

There is a problem with characterizing the fault tolerance of a network, namely, the number of possible fault combinations grows explosively. Thus, only for very small networks can every possible fault pattern be investigated. This dissertation has focused on a method which prevents the need to evaluate every possible fault combination, and still provide a reasonable estimate of the worst case network performance. This method is a worst case path approach.

Faults are assumed to occur one at a time, and the fault which occurs is the one which causes the maximum performance degradation at that step. Thus, the first fault to occur is the single fault which causes the maximum degradation in performance. The second fault is the single fault which causes the maximum incremental performance degradation in the network in which the first fault has already occurred, and so forth.

Additionally, a performance measure was proposed which allows networks trained to different criteria to be compared. This measure, called approximation quality (AQ), compares the RMS approximation error to the RMS value of the function to be learned. Coupling this measure with the worst case path approach provides a powerful, flexible, and relatively efficient method for quantitatively measuring the fault tolerance of a network, and for making reasonable comparisons of the relative fault tolerance of different networks.

Although there are a number of possible fault types that can occur in a network, this dissertation has concentrated primarily on the loss of a weight (or equivalently a connection). Some work was also done to investigate the partial fault tolerance, i.e., when a weight is not lost completely, but is altered by $\pm 10\%$. The partial fault tolerance may be of particular interest in networks where weight precision is limited and where weight values may change with time due to aging or noise. [Kramer, 1991]

The fault tolerance of the multilayer perceptron was investigated. Three different training methods were evaluated. It was found that the training method had a significant impact on the quality of the solution found (in terms of fault-free approximation error) and on the magnitude of the degradation caused by the worst single fault. However, it was also found that the multilayer perceptron was highly lacking in fault tolerance. Even with the best training method, the worst single fault caused an increase in RMS error which was approximately twice the RMS value of the function to be learned. As the worst case path was followed, the approximation error increased to more than twenty times the function's RMS value!

The fault tolerance of the Gaussian radial basis function network was also tested. It was found that this network not only learned faster than the multilayer perceptron, but also was much more fault tolerant. Indeed, as the worst case path was followed, the RMS approximation error never exceeded the function RMS value, even in the presence of an extremely large number of faults. This is a highly desirable property, since regardless of the number of faults, the network is still better than no network (unlike the multilayer perceptron, where a single fault can effectively make the network worse than the total absence of the network).

A neural network can often be thought of as performing function synthesis using linear combinations of (potentially) non-orthogonal basis functions. Spectral techniques may be employed as an aid in choosing network parameters which give good learning performance and good fault tolerance. The spectrum of the sigmoid function is ill-behaved and thus is unlikely to give good behavior on most learning tasks. The Gaussian function, on the other hand, has a well behaved spectrum which may be easily controlled by the Gaussian width parameter.

Spectral techniques may also be employed to explain why a network doesn't perform well, such as in the case of CMAC slow learning. Additionally, it was shown that when a Gaussian activation function was used which had a strong spectral mismatch with the function to be learned, the learning speed and fault tolerance of the network degraded to a level comparable to that of the multilayer perceptron.

There are numerous ways to improve the fault tolerance of a network. The effective utilization of spectral information is one of them. Another possible means of fault tolerance enhancement is engineered redundancy. This method may be applied to weights, processing elements, or even entire networks. The philosophy of the method is to simply spread functionality across as many physical elements as necessary to achieve the desired fault tolerance. Additionally, well chosen initial conditions for training can improve fault tolerance.

A training method due to Séquin and Clay was shown to significantly improve the fault tolerance of a network. This method randomly chooses some (fixed) number of nodes to be disabled at each point in training. The network is then trained without these nodes present, and then the nodes are re-enabled. At the next training point a new set of randomly chosen nodes are disabled and the process is repeated. This method was tested on both the multilayer perceptron and the Gaussian radial basis function network. Both networks showed an increase in fault tolerance at the expense of greater fault-free approximation error. The Gaussian radial basis function network, however, showed a much greater improvement in fault tolerance and a much smaller increase in fault-free approximation error than did the multilayer perceptron. Indeed, the Gaussian radial basis function network reached a state where the loss of the most critical network weight produced essentially no performance degradation.

Still another way to improve the fault tolerance of a network is to limit the effect that a fault could have. This could be achieved by enforcing limited dynamic range weights (thus limiting the possible fault magnitude), by training continuously to compensate for any faults which may occur, and by limiting connectivity in the network to prevent the effect of the fault from traveling throughout the entire network.

Finally, methods for choosing network parameters for improved fault tolerance were discussed. Perhaps the most important choice is the activation function. The activation function should be chosen to be spectrally similar to the function to be learned. Furthermore, the activation function should be chosen to be as wide as possible (subject to the constraint of a good spectral fit), so that the space of interest may be spanned with a reasonable number of processing elements and so that a reasonable number of training points may be used. The initial centers or biases should be chosen so that the processing elements are uniformly distributed throughout the space of interest (unless *a priori* knowledge dictates that some other placement would be better). Network weights should be initialized to small, safe values. Zero valued weights in the output layer and small random weights in any other layers are usually good initial choices.

8.2 Possible Future Work

This dissertation has opened up a great many areas for exploration. For instance, using the AQ measure and the worst case fault path is a very general method of measuring network fault tolerance. Furthermore, this method allows the comparison of networks trained to different criteria (or the same network trained to different criteria). This dissertation has focused on the multilayer perceptron and the Gaussian radial basis function network (with Gaussian activation functions) trained to approximate a sinusoidal function. It would be highly desirable to characterize other networks for this task. It also would be highly desirable to characterize these networks at other tasks.

It was found that training with intermittent faults improved the fault tolerance (compared to ordinary gradient descent) of both the multilayer perceptron and the Gaussian radial basis function network. It is natural to ask how other training methods affect fault tolerance and if training methods might be found which perform better still.

The Gaussian activation function was found to have a frequency domain representation that was well-behaved. The sigmoid activation has a frequency domain representation which is ill-behaved. There are a number of functions which have interesting spectral properties which may prove useful as activation functions. For instance, the sinc function has a spectrum which is a rectangular pulse. Such an activation function could have interesting properties in terms of fault tolerance, and in which functions may or may not be learned, since the frequency content is constant over some band and zero elsewhere.

Another potentially interesting class of basis functions are wavelets or frames. These have spectra which are bandpass in nature. Of course, the ultimate bandpass activation function is a sinusoid. All of these are relatively unexplored.

This dissertation has dealt primarily with faults in the form of the loss of a network weight (or equivalently a network connection). There are of course other fault models which could be studied [Bolt, 1991]. It would be informative to identify the possible faults in a hardware implementation of a neural network and study the effect of these specific types on network performance.

Finally, all of the experiments discussed in this dissertation have dealt with single input, single output networks having a single layer of processing elements, however, the methods presented here are not limited to this type of network. Quite to the contrary, these methods extend naturally to more complex networks. Furthermore, it is likely that extending these methods will provide further insights into the general properties of artificial neural networks.

REFERENCES

[Albus, 1971]	Albus, J.S., "A Theory of Cerebellar Functions", *Mathematical Biosciences*, Vol. 10, pp. 25-61, 1971
[An, 1991]	An, P.E., "An Improved Multi-Dimensional CMAC Neural Network: Receptive Field Function and Placement", PhD Dissertation, University of New Hampshire, 1991
[Bolt, 1991]	Bolt, G., "Investigating Fault Tolerance in Artificial Neural Networks", University of York, Department of Computer Science, Technical Report YCS 154, Heslington, York, England, 1991
[Bolt, 1991A]	Bolt, G., Personal Communication, 1991
[Carlson, 1986]	Carlson, A.B., *Communication Systems-An Introduction to Signals and Noise in Electrical Communication*, Third edition, New York, NY, McGraw-Hill Book Company, 1986
[Carrara, 1992]	Carrara, D., "Investigating and Improving the Fault Tolerance of the Multilayer

Perceptron", Master's Thesis (In Preparation), University of New Hampshire, 1992

[Carter, 1988] Carter, M.J., "The Illusion of Fault Tolerance in Neural Networks for Pattern Recognition and Signal Processing", *Proceedings Technical Session on Fault-Tolerant Integrated Systems*, University of New Hampshire, Durham, NH, 1988

[Carter, 1990] Carter, M.J., Rudolph, F.J., Nucci, A.J., "Operational Fault Tolerance of CMAC Networks", *Advances in Neural Information Processing Systems 2*, D.S. Touretzky (Ed.), San Mateo, CA: Morgan-Kaufmann Publishing Co, 1990

[Carter, 1990A] Carter, M.J., Nucci, A.J.,Miller, W.T., An, P.E., Rudolph, F.J., "Slow Learning in CMAC Networks and Implications for Fault Tolerance", *Proceedings 24th Annual Conference on Information Sciences and Systems*, Princeton University, March 1990

[Clay, 1991] Clay, R.D., and Séquin, C.H., "Limiting Fault-Induced Output Errors in ANN's", *Proceedings International Joint Conference on Neural Networks*, Seattle, WA, 1991

[Darken, 1990] Darken, C., and Moody, J., "Fast Adaptive K-Means Clustering: Some Empirical Results", *Proceedings International Joint Conference on Neural Networks*, pp. II-233-238, San Diego, CA, June, 1990

[Fahlman, 1990] Fahlman, S.E., and Lebiere, C., "The Cascade-Correlation Learning Architecture", *Neural Information Processing Systems 2*, Touretzky, D.S. (Ed.), San Mateo, CA, Morgan-Kaufmann Publishing Co., pp. 524-532, 1990

[Fahlman, 1988] Fahlman, S.E., "Faster-Learning Variations on Back-Propagation: An Empirical Study", *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, CA, Morgan-Kaufmann Publishing Co., 1988

[Holmes, 1991] Holmes, E.F., "Operational Fault Tolerance and Fault Recovery of a One-Dimensional CMAC Neural Network", Master's Thesis, University of New Hampshire, 1991

[Izui, 1990] Izui, Y., and Pentland, A., "Analysis of Neural Networks with Redundancy", Neural Computation, Vol. 2, Number 2, pp. 226-238, Summer, 1990

[Johansson, 1990] Johansson, E.M., Dowla, F.U., and Goodman, D.M., "Backpropagation Learning for Multi-Layer Feed-Forward Neural Networks Using the Conjugate Gradient Method", Lawrence Livermore National Laboratory, Preprint UCCL-JC-104850, 1990

[Jordan, 1988] Jordan, M.I., "Supervised learning and systems with excess degrees of freedom", COINS Technical Report 88-27, Massachusetts Institute of Technology, May 1988

[Kleinfeld, 1989] Kleinfeld, D., and Sompolinsky, H., "Associative Network Models for Central Pattern Generators", Methods in Neuronal Modeling: From Synapses to Networks, Koch, C., and Segev, I., (Eds.), Cambridge, MA, MIT press, pp. 195-246, 1989

[Kramer, 1991] Kramer, A., Sin, C. K., Chu, R., Ko, P. K., "Compact EEPROM-based Weight Functions", *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S. Touretzky, (Eds.), San Mateo, CA: Morgan-Kaufmann Publishing Co, 1991

[Kruschke, 1989] Kruschke, J.K., "Distributed Bottlenecks for Improved Generalization in Backpropagation Networks", *Neural Networks*, Vol. 1, No. 3, pp. 187-193, July, 1989

[Kruglyak, 1991] Kruglyak, L., and Bialek, W., "Analog Computation at a Critical Point: A Novel Function for Neuronal Oscillations?", *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S. Touretzky, (Eds.), San Mateo, CA: Morgan-Kaufmann Publishing Co, 1991

[LeCun, 1990] LeCun,Y., Denker,J.S., and Solla,S.A., "Optimal Brain Damage", *Advances in Neural Information Processing Systems 2*, D. S. Touretzky (Ed.), San Mateo, CA: Morgan-Kaufmann Publishing Co, 1990

[Lippmann, 1988] Lippmann, R.P., "An Introduction to Computing with Neural Nets", (reprinted from IEEE ASSP, pp. 4-22, April, 1987), *Artificial Neural Networks: Theoretical Concepts*, V.Vemuri (Ed.), IEEE Computer Society Press, pp. 36-54, Washington, D.C., 1988

[McCarthy, 1990] McCarthy, R.A., and Warrington, E.K., *Cognitive Neuropsychology A Clinical Introduction*, Academic Press, San Diego, CA, 1990

[Miller, 1990]	Miller, W.T., Glanz, F.H., and Kraft, L.G., "CMAC: An Associative Neural Network Alternative to Backpropagation", *Proceedings of the IEEE*, Vol. 78, pp. 1561-1567, October, 1990
[Miller, 1991]	Miller, W.T., Box, B.A., Whitney, E.C., Glynn, J.M., "Design and Implementation of a High Speed CMAC Neural Network Using Programmable CMOS Logic Cell Arrays", *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S. Touretzky, (eds.), San Mateo, CA: Morgan-Kaufmann Publishing Co, 1991
[Moody, 1991]	Moody, J.E., "Note on Generalization, Regularization, and Architecture Selection in Nonlinear Learning Systems", *Proceedings IEEE Conference on Neural Networks and Signal Processing*, Princeton, NJ, September, 1991
[Mozer, 1989]	Mozer, M. C. and Smolensky, P. "Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment", Department of Computer Science & Institute of Cognitive Science,

University of Colorado, Boulder, Technical Report CU-CS-421-89, January, 1989

[Nelson, 1991] Nelson, M. M., and Illingsworth, W.T., *A Practical Guide to Neural Nets*, Addison-Wesley Publishing Company, Reading, MA, 1991

[Neti, 1990] Neti, C., Schneider, M., and Young, E., "Maximally fault tolerant neural networks and nonlinear programming", *Proceedings International Joint Conference on Neural Networks,* pp. II-483-496, San Diego, CA, June, 1990

[Nguyen, 1990] Nguyen, D., and Widrow, B., "The truck backer-upper: An example of self-learning in neural networks", *Neural Networks for Robotic Control*, W.T. Miller, R. Sutton, P. Werbos (eds.), MIT Press, Cambridge, MA, 1990

[Niebur, 1991] Niebur, E., Kammen, D.M., Koch, C., Ruderman, D., Schuster, H.G., "Phase-coupling in Two-Dimensional Networks of Interacting Oscillators", *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody and D. S.

Touretzky, (eds.), San Mateo, CA: Morgan-Kaufmann Publishing Co, 1991

[Parker, 1985] Parker, D.B., "Learning Logic", Technical Report TR-47, Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science, Cambridge, MA, 1985

[Pati, 1990] Pati, Y.C., and Krishnaprasad, P.S., "Analysis and Synthesis of Feedforward Neural Networks Using Discrete Affine Wavelet Transformations", University of Maryland, Electrical Engineering Department and Systems Research Center Technical Report TR 90-44, 1990

[Phatak, 1990] Phatak, D.S., and Koren, I., "A Study of Fault Tolerance Properties of Artificial Neural Nets", Technical Report, Electrical and Computer Engineering Department, University of Massachusetts, Amherst, MA, 1990

[Poggio, 1990] Poggio, T. and Girosi, F., "Networks for Approximation and Learning", *Proceedings of the IEEE*, Vol. 78, pp. 1481-1497, September, 1990

[Rumelhart, 1986] Rumelhart, D.E., and McClelland, J.L, (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1, Chapter 8, Cambridge, MA, MIT Press, 1986

[Rumelhart, 1988] Rumelhart,D.E., "Learning and Generalization", *Proceedings IEEE International Conference on Neural Networks*, San Diego, 1988 (plenary address)

[Sanner, 1991] Sanner, R.M., and Slotine, J.E., "Gaussian Networks for Direct Adaptive Control", Nonlinear Systems Laboratory Technical Report NSL-910503, Massachusetts Institute of Technology, May, 1991

[Segee, 1991] Segee, B.E., and Carter, M.J., "Fault Tolerance of Pruned Multilayer Networks", *Proceedings International Joint Conference on Neural Networks*, pp. II-447-452, Seattle, WA, 1991

[Segee, 1990] Segee, B.E. and Carter, M.J., "Fault Sensitivity and Nodal Relevance Relationships in Multi-Layer Perceptrons", UNH Intelligent Structures Group Report ECE.IS.90.02, March 9, 1990

[Séquin, 1990] Séquin, C.H., and Clay, R.D., "Fault Tolerance in Artificial Neural Networks", *Proceedings International Joint Conference on Neural Networks*, pp. I-703-708, San Diego, CA, June, 1990

[Siewiorek, 1991] Siewiorek, D.P., "Architectures of Fault-Tolerant Computers: An Historical Perspective", Proceedings of the IEEE, Vol. 79, No. 12, pp. 1710-1734, December, 1991

[Strang, 1989] Strang, G., "Wavelets and Dilation Equations: A Brief Introduction", *SIAM Review*, Vol. 31, No. 4, pp. 614-627, December, 1989

[Vemuri, 1988] Vemuri, V., "Artificial Neural Networks: An Introduction", *Artificial Neural Networks: Theoretical Concepts*, V.Vemuri (Ed.), p 2, Washington, D.C., IEEE Computer Society Press, 1988

[Watrous, 1987] Watrous, R.L., "Learning Algorithms For Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization", *Proceedings IEEE First International Conference on Neural Networks*, Volume II, pp. 619-627, San Diego, CA, 1987

[Werbos, 1974]	Werbos, P.J., "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences", PhD Dissertation, Harvard University, 1974
[Werbos, 1990]	Werbos, P.J., "Backpropagation through time: What it is and how to do it", *Proceedings of the IEEE*, Vol. 78, pp. 1550-1560, October, 1990