Using The Modified Back-propagation Algorithm To Perform Automated Downlink Analysis

by

Nancy Y. Xiao

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Author .....
Depar ..... puter Science
May 28, 1996

Certified by. .....
rnard C. Lesieutre
trical Engineering
Thesis Supervisor

Accepted by .....
F.R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Using The Modified Back-propagation Algorithm To Perform Automated Downlink Analysis

by

Nancy Y. Xiao

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1996, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A multi-layer neural network computer program was developed to perform supervised learning tasks. The weights in the neural network were found using the back-propagation algorithm. Several modifications to this algorithm were also implemented to accelerate error convergence and optimize search procedures.

This neural network was used mainly to perform pattern recognition tasks. As an initial test of the system, a controlled classical pattern recognition experiment was conducted using the X and Y coordinates of the data points from two to five possibly overlapping Gaussian distributions, each with a different mean and variance. The objective of this experiment was to determine the distribution to which the data point belonged.

The neural network was then trained using data from the Mars Observer Ka-Band Link Experiment in an attempt to learn to detect anomalous operations in NASA's Deep Space Network's (DSN) downlink system.

Thesis Supervisor: Bernard C. Lesieutre
Title: Assistant Professor of Electrical Engineering

Company Supervisor: Stephen Townes
Title: Manager Space Communications Technology Program, Jet Propulsion Laboratory

Acknowledgments

First, I would like to thank my JPL supervisor, Dr. Steve Townes, for giving me the opportunity to work on this challenging and interesting project. Also, his guidance and wisdom have helped me to mature as a scientific researcher. I am also very grateful to my JPL colleagues, Dave Watola and John Hampshire, for their technical expertises and constant guidances. I must also thank my MIT advisor, Prof. Lesieutre, for his enthusiasm, creative thinking and generous supervisions.

I am also forever grateful to Linda Rodgers and all the people working in the JPL co-op program for making my time working at JPL a truly fun and wonderful experience.

For this opportunity, I must also thank all of my friends. I am so lucky to have met all of you. Your kind offerings of ideas and support, your ingenious imaginations and talents, and your sincerities and affections have touched me so deep. Your friendships have shaped my life in ways I never thought were possible. I will forever cherish all the wonderful memories I have shared with each of you; and I hope that we will soon meet again.

Finally, to my family: Mom, Dad, and Joe, and to my Grandparents, thank you for believing in me and for making all the sacrifices to help me to become the best I can be.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Jet Propulsion Laboratory (JPL) operates the Deep Space Network (DSN), which is NASA's communication link to all unmanned spacecraft operating in the solar system. The Deep Space Stations (DSS) and all the associated telemetry equipment at DSN facilities are of critical importance for tracking planetary missions and receiving telemetry signals from spacecrafts. Currently, these Deep Space Stations are being monitored by JPL engineers and operations personnel to detect and correct any system failures that might cause irrecoverable data loss. However, planetary missions have become significantly more demanding, and constant manual supervision is extremely inefficient and costly. Therefore, there is an urgent need for an automated health monitoring system that is capable of performing real-time detection and diagnosis of anomalous operations in the DSN. The Downlink Analyzer (DLA), a hybrid learning and monitoring system, is designed to accomplish this complex task. [4]

The purposes of this thesis are to show that a neural network based model is capable of recognizing the complex patterns associated with the downlink system's health states, and to study some of the issues surrounding the successful implementation of such a neural network model. Several reports on the application of neural networks for fault diagnosis have appeared in the literature [13] [1] [2].

Chapter 1 describes the basic overview of the downlink analyzer, and a description of the DLA interface, the monitoring phase of the system. Chapter 2 describes the fundamentals of neural network. Chapter 3 describes the back-propagation algorithm,

its application in supervised learning and the its implementation using MATLAB. Chapter 4 describes the series of tests that were conducted to test the reliability of my neural network software. Chapter 5 describes some of the experiments conducted to study the effects of input space topology and of network architecture on the network's performance. Finally, Chapter 6 describes the status of the neural network based Downlink Analyzer and concludes with suggestions for future work.

1.1 Overview of the Automated Downlink Analyzer

The Downlink Analyzer is intended to be a real-time operations tool that monitors the DSN side of the downlink channel to detect faults of the downlink telemetry and equipment. It could also be used as a near-real-time time series analysis tool that helps operators, engineers and scientists make detailed quantitative assessments. Figure 1-1 is a brief overview of how the Downlink Analyzer might operate and interface with the Fault Detection, Isolation, and Recovery subsystem (FDIR) [4].

Inside the Downlink box is a system-level representation of a typical DSN ground station that can be broken into several subsystems: antenna, microwave, receiver, and decoder. Data collected from these subsystems, plus the total power radiometer (TPR), which provides the noise temperature estimates, *No*, are the information-bearing signals known as the causal factors [9]. These factors are used by the neural network model-based Downlink Analyzer to estimate the health state of the downlink system, which can be compared to the system's actual health state, represented by carrier-to-noise ratio estimate, *Pc/No*, which is denoted as the Principal Health Metric. A detailed explanation of these signals and their usage is given in Chapter 5.

1.1.1 Fault Detection

First, signals that characterize the state of downlink sub-systems or critical components within those subsystems are fed in as input feature vectors. Then, a selected

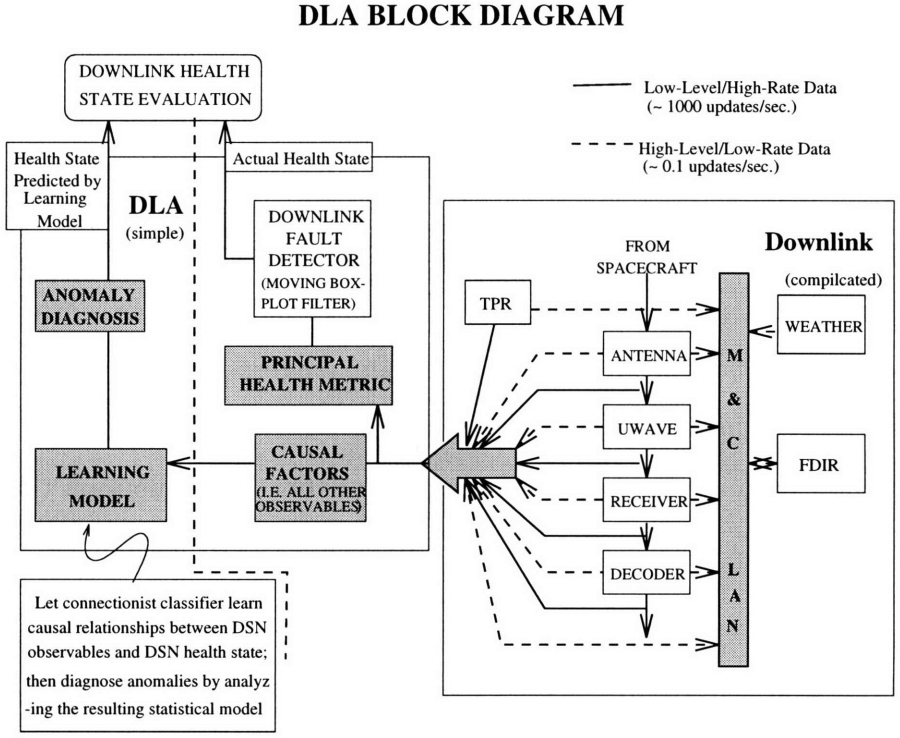

Figure 1-1: Automatic Downlink Analyzer

subset of system observables, signals strongly correlated with the health state of the downlink, are denoted as principal health metrics. They are analyzed by a moving boxplot filter. A moving boxplot filter is a nonlinear smoothing filter which takes in any specified number of anti-casual and casual data points, and outputs the median value. After the principal health metrics have been smoothed by the moving boxplot filter, it is then passed into the Downlink Fault Detector, which classifies the downlink "health state" into four possible categories: very good, good, bad, and very bad [4].

1.1.2 Fault Diagnosis

Once a downlink fault is detected, meaning the downlink's "health state" is classified as either bad or very bad, the DLA will initiate a detailed diagnostic procedure to determine which combination of inputs is most likely to have caused the detected fault. There are various technical approaches for building a fault diagnostic model. The ultimate objective of the DLA is to perform fault detection on any general system

that is comprised of subsystems for which no precise parametric model is available, and no prior knowledge is assumed. Under such circumstances, a neural network is a good choice for the DLA fault diagnostic model.

In the DLA learning model, the neural network learns the causal relationships between the downlink observables and the downlink health state by pairing the neural computation output associated with each input pattern with a target vector representing the actual heath state of the downlink system, and then produce a set of optimal weight parameters that minimizes a pre-defined cost function that represents a measure of the difference between the neural network's output and the target vector.

Once the neural network learning model has been trained to classify the health state of the system, and if the system's health state is classified to be bad, then an analytical procedure based on a Taylor Series approximation is used to determine the possible cause of the anomaly. Since the neural network gives a one-to-one mapping from the feature vector space to the classification space, or health state space, by inverting this mapping one can determine the cause of an anomaly. The cost function, or objective function,reflects the empirical relationship between the input patterns and their corresponding class membership. Thus, we can estimate how likely it is for an input factor to be responsible for the system's fault by changing the factor's value from its false value to its healthy state value. If this change causes a significant change in the objective function,then we can conclude that this particular input factor is very likely to have caused the system's anomaly. If it did not induce any signif

icant change in the objective function, then it is probably not a cause. This parameter, which is defined as the change in the objective function caused by changing an input factor from its false to its normal value, is known as the saliency of an input factor. The higher the factor's saliency, the more likely it is to be responsible for the anomaly [7].

1.2 DLA Graphical User Interface

The front end of the Downlink Analyzer is a graphical user interface designed by Dave Watola at the Jet Propulsion Laboratory. It is used to display input data, compute the "health state" for the system, and output files of selected feature extractions. A sample DLA graphical user interface screen is shown in Figure 1-2.

1.2.1 Visible Features

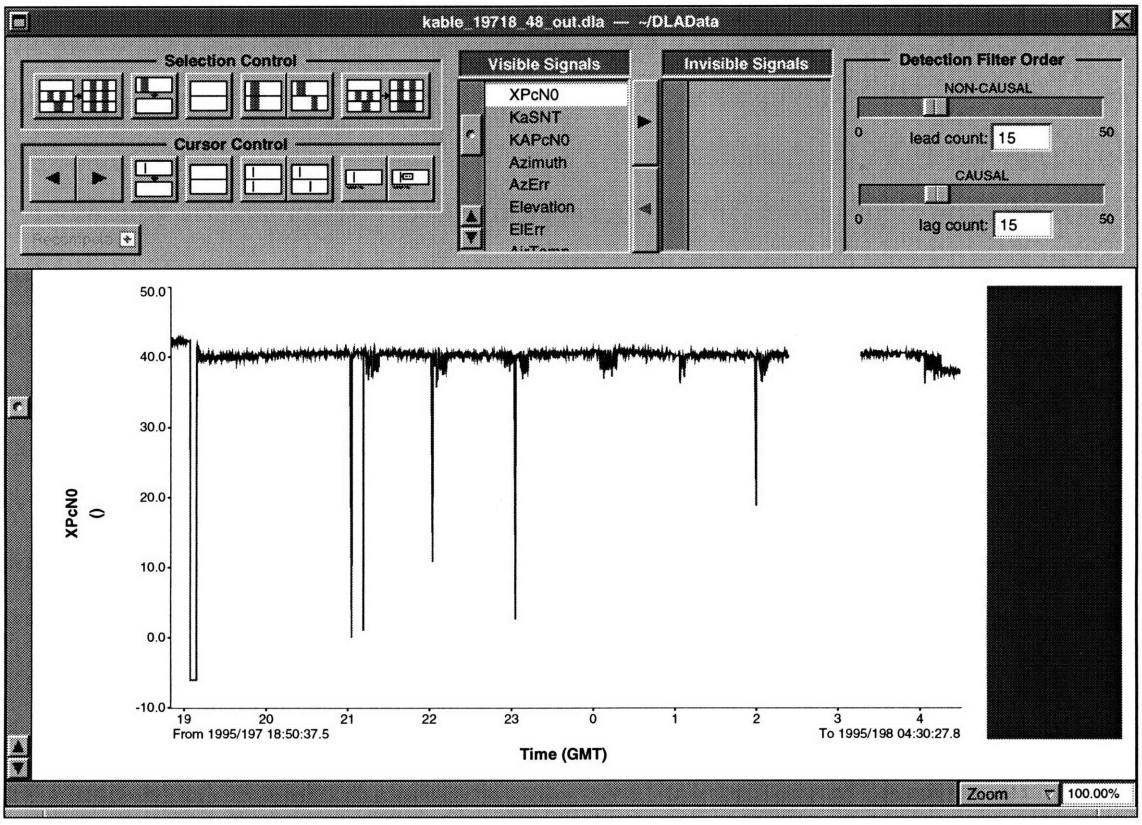

Figure 1-2: DLA Interface

On the top left corner, the **Selection Control** and **Cursor Control** buttons are responsible for Copy, Paste, Merge, Move, or Clear selected signals. All these functions can be done in two different modes. In a **Tied Selections** mode, a change made in the selections of the current signal will also be made in all other signals in the document, while in a **Untied Selections** mood, a change will only affect the current signal.

On the top middle portion, the **Visible Signals** window are where a list of signals is displayed in the same order as in the scrollable graphic area. Principal health metrics are flagged by a red cross to the left of their names. Double clicking on a signal's name in the **Visible Signals** window will cause its graph to become visible (i.e., displayed in the scroll view), and it also makes it the currently focused signal, while single-clicking on it gives it the current focus but without the scrolling view. A signal in the **Invisible Signals** window is unseen from the graphic area. Double clicking on it moves it back to the bottom of the visible list and makes it the currently focused Signal. Signals can be transferred between the visible and invisible window using either the move left or move right button. These allow for controlled arrangement of the input signals.

On the top right corner, in the **Detection Filter Order** area is where one can set the size of the boxplot detection filter by adjusting the **NON-CAUSAl** and **CAUSAL** sliders. After these are set to the desired numbers, pressing the **Recompute** button will calculate and display Health Metric Classifications. Health Metric Classifications are only displayed on top of a Principal Health Metrics signal in graphic area. The various colors indicate the different "health states." Also, in the graphic area is the **Boxplot View** which is on the right of the signal display. Clicking there displays information on boxplot and statistics and shows health state colors for Principal Health Metrics signals, or a solid blue background for all other signals.

On the buttom of the graphic area (i.e., underneath the displayed signal), the timetag of both the left and rightmost visible data point. One can also use the cursor to high-light specific regions in the data; both the cursor position and data value are shown. Zoom options are also provided.

1.2.2 Feature Vector Extraction

Feature Vector Extraction is a very important data preprocessing procedure, whose function is to extract from available data features that appear most helpful for classification purposes. The performance of any pattern recognition system is highly

correlated with the quality of the feature selection process. The DLA interface is equipped with an comprehensive Interactive Feature Vector Extraction tools.

First, a **.feature** file should be created, in which one can specify the names of signals from which to extract features, and what kind of features to extract. Available features include but are not limited to the following: the original signal, its mean, variance, minimum and maximum data values, and any algebraic and trigonometric operations.

After a desired set of features have been specified in a .feature file, one simply needs to select an input file to use as a data source and to name an output file in which the result of the feature extraction will be stored.

Chapter 2

Fundamentals of Neural Network

Much of the inspiration for neural network models comes from neuroscience, and the network model exhibits a number of the brain's characteristics. The three most essential characteristics are its abilities to learn, to generalize and to abstract pertinent information. Some neural networks learn from experience; they are able to self-adjust to produce consistent outputs when given a set of inputs. This training process is known as unsupervised learning. Another type of learning process, called supervised learning, occurs when a set of desired outputs is presented together with the input set. Once a network is trained, it is, to a certain degree, insensitive to minor variations in its test inputs. In other words, it is able to interpolate and extrapolate from the training examples to generalize to new situations. In order to construct such relationships, a neural network needs to discern and recognize patterns buried in noise and distortion. Finally, a neural network can extract an idealized prototype from distorted inputs [12].

Despite these brainlike capabilities, neural networks are still far away from mimicking the complex and difficult physiological and psychological functioning of the human nervous system. In fact, the structure of a neural network bears only a superficial resemblance to the brain's communications system. However, understanding the human neural system, and producing a computational system that performs brainlike functions are two mutually reinforcing scientific researches that will continue to make significant progress as technology matures [12].

2.1 Historical Background

The history for computational or neural modeling can be traced back to the early 1940s. However, the philosophical and psychological perspectives of such ideas were originated and studied by great thinkers like Plato and Aristole [12].

The first important paper on neural network was published in 1943, by McCulloch and Pitts. They proposed a simple model of a neuron that produces either a 1 or 0, depending on whether or not the weighted sum of its inputs exceeds a set threshold (a schematic diagram of a neuron and explanations of its functionality appears in the next section). In 1949, D. O. Hebb proposed a learning law that became the starting point for neural network training algorithms [12].

In the 1950s and 1960s the first neural networks were produced. Initially they were implemented in hardware, and later they were converted to software simulations. The first type of neural network was often called Perceptrons. It is a network consisting of a single layer of artificial neurons developed by Marvin Minsky, Frank Rosenblatt and Bernard Widrow. In 1962, Rosenblatt was able to prove the convergence of a learning algorithm, a way to adjust the weights iteratively to obtain a set of desired outputs. This development advocated a great deal of enthusiasm and hope in the artificial intelligence community. However, a few years later Minsky and Papert pointed out in their book Perceptrons [8] that the single layered networks were theoretically incapable of solving many simple problems, such as the simple exclusive or (XOR) problem, which is the linear separability limitations associated with single layer networks. Although Rosenblatt believed that overcoming such limitations can be accomplished using multi-layered neural networks, there was no learning algorithm known which could calculate the weights necessary to implement a given computation. Minsky and Papert were not optimistic about the potential for finding a theoretically sound algorithm for training multilayer neural networks. Minsky's book persuaded many discouraged researchers to leave the field; the neural network paradigm was left in an virtual stagnation for almost two decades [12].

The most influential development happened around 1985, when various researchers

almost simultaneously invented a systematic method for training multilayer neural network, known as back-propagation. It appears that it was first invented by Werbos in 1974, and almost ten years later it was rediscovered independently by Rumelhart, Hilton and Williams. The discovery of the back-propagation algorithm has dramatically expanded the range of problems to which neural network can be applied, and many current activities are centered on back-propagation and its variations [12].

2.2 Inspiration from Neuroscience

Neural networks are biologically inspired; they are an extremely simplified version of the astonishingly complex human nervous system. nevertheless, they provide important insights in understanding the collective behavior of a network of cells and the powerful potential of neural computing [12].

2.2.1 Biological Neurons

The brain is composed of about 10^{11} neurons and close to 10^{15} inter-connections among the many different type of neurons [12].

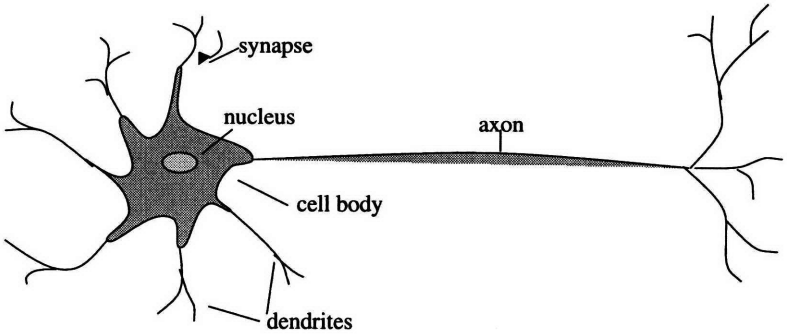

Figure 2-1: Schematic Drawing of a Biological Neuron

A schematic drawing of a biological neuron is shown in Figure 2-1. Each neuron is comprised of five basic components: the cell body or soma, nucleus, dendrites, synapse, and axon. Dendrites are branchlike nerve fibers connected to the cell body. They are responsible for receiving signals from a connection point called a synapse. A synaptic junction has both a receiving and transmitting side. Once a signal is received, it is then transmitted through a complicated chemical process in which specific transmitter substances are released from the sending side of the synaptic junction, in turn changing the electrical potential inside the cell body of the receiving neuron. If this potential exceeds a threshold, a pulse of set duration and strength would be "fired" down the axon to the other neurons. An artificial neuron is designed to model these simple characteristics of a biological neuron [12].

2.2.2 Artificial Neurons

An artificial neuron is designed to mimic the basic functions of a biological neuron. Specifically, the artificial neuron computes the weighted sum of its applied inputs; each input represents the output of another neuron. The calculated sum is analogous to the electrical potential of a biological neuron. The output is then passed through an activation function, which determines whether or not it has exceeded a set threshold value, T. Figure 2-2 shows a neuron model that implements this idea [12].

Here, each input to the neuron is labeled $x_1, x_2, \ldots, x_n$, collectively they are referred to as the input vector X. Each input is multiplied by a corresponding weight $w_1, w_2, \ldots, w_n$ similar to the synaptic strength in a biological neuron. Weighted inputs are applied to the summation block, labeled $\sum$. This summation block functions like a cell body, sum all the weighted inputs, and produces an output called NET [12].

$$NET = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

This same equation can be stated in a vector notation as follows:

$$NET = XW$$

Neural Network Diagram

- **Single Artificial Neuron**

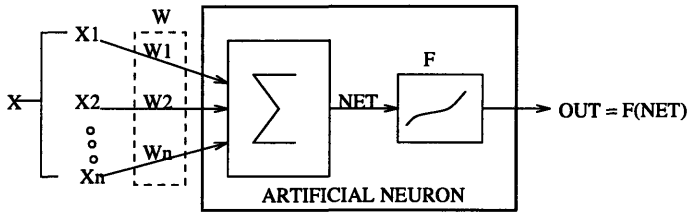

Figure 2-2: Schematic Diagram of an Artificial Neuron

The NET signal is then further processed by an activation function called $\mathcal{F}$. This activation function can be as simple as a binary threshold unit, where,

$$OUT = \mathcal{F}(NET)$$

and,

$$OUT = 1 \text{ if NET} > T$$

$$OUT = 0 \text{ otherwise}$$

However, a more general activation function that is often used in neural networks is called a squashing function, or sigmoidal logistic function. This function is mathematically expressed as $\mathcal{F}(x) = 1/(1 + e^{-x})$, thus in the artificial neuron model,

$$OUT = 1/(1 + e^{-NET})$$

This sigmoidal logistic function is illustrated in the Figure 2-3.

There are several reasons for choosing the squashing function to be the activation function. First of all, its S shape provides appropriate gain for a wide range of input levels. This nonlinear gain is calculated by finding the ratio of the change in OUT to a small change in NET, and this calculation is equivalent to taking the derivative

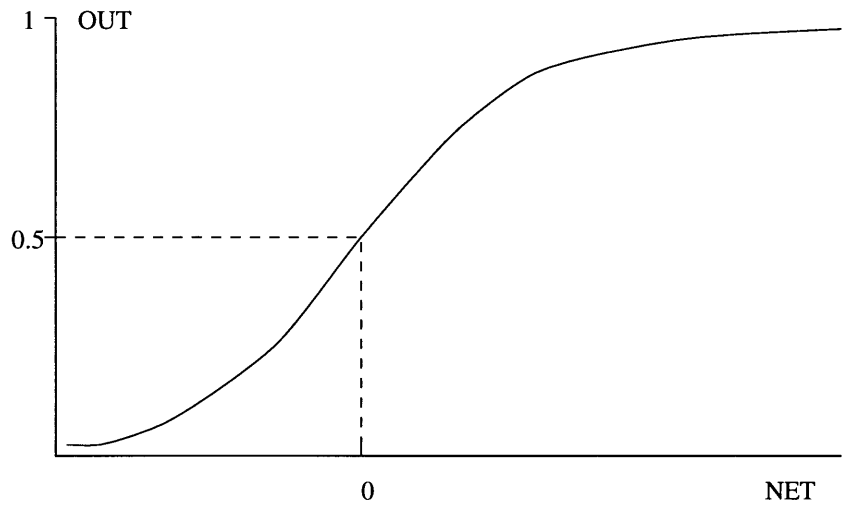

Figure 2-3: Sigmoidal Logistic Function

of the squashing function (i.e., $\mathcal{F}(NET)$) with respect to NET. We can see that at extremely large positive and negative input levels the gain is very small so to avoid saturation. While around the central region input levels are small (NET is near zero) and the steep slope of the squashing function producing a high gain to amplify these small signals. Besides its ability to provide automatic gain control, the squashing function also has other desirable properties: it is differentiable everywhere and its derivative is very simple to calculate; in fact, it can be expressed in terms of the function itself.

$$\frac{\partial \text{ OUT}}{\partial \text{ NET}} = OUT(1 - OUT) \tag{2.1}$$

Another commonly used activation function is the hyperbolic tangent function,

$$OUT = tanh(x)$$

This function has a bipolar value for OUT, which is beneficial for certain network architectures.

This concludes the description of an artificial neuron. Each neuron has limited functionality, capturing some of the essential characteristics of a biological neuron.

A mutilayered network of these individual neurons is powerful enough to execute complicated programs in a robust manner. This leads to the next topic: Parallel Processing.

2.3 Parallel Processing

Each biological neuron in the brain is a processor which is executes a very simple program: it computes the weighted sum of input data, which are outputs of other processors, and then outputs a single number, which is a nonlinear function of this weighted sum. The brain can be described as a parallel system of about 10^{11} of such processors, where the output of one processor is sent to other processors, which are executing the same kind of computation. Different processors are using different weights and possibly different activation functions. Figure 2-4 is an example of a two-layer neural network [5].

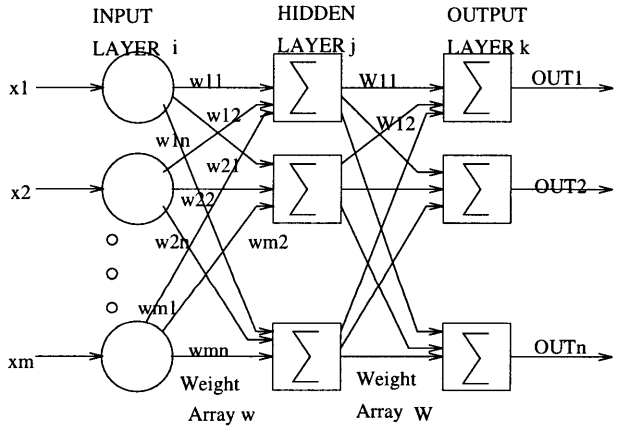

Figure 2-4: A Two-layer Neural Network

One remarkable property associated with parallel processing is its robustness. In an ordinary sequential computation, one or at most a few processors execute very complicate programs; and a single bit error may easily ruin the entire computation.

However, in a parallel system such as a multilayer neural network, each processing unit works independently, and the result depends on a sum of many terms as shown in Equation 2.2. Thus, errors in just a few of the terms will probably be inconsequential.

$$OUT_i = F(\sum_k W_{jk}(\sum_j w_{ij}x_i)) \tag{2.2}$$

2.4 Back-propagation

Back-propagation is a powerful tool. Its invention in 1985, played an important role in resurging interest in neural networks. Prior to its invention, there was no known training algorithm for multilayer networks, and single layer perceptrons can only perform linear separable functions [12].

Back-propagation is an efficient and simple method for calculating exact derivatives of a single target quantity with respect to a large set of input parameters. It can be applied to a variety of problems, such as pattern recognition, fault diagnosis, and dynamic modeling, and to almost any system built up from elementary subsystems, such as a neural network built up from artificial neurons, with the restriction that each subsystem must be both continuous and differentiable functions that are known to the users [16].

2.4.1 Use of Back-propagation to Perform Supervised Learning Tasks

Back-propagation is a popular method used to perform supervised training, which is symbolized in Figure 2-5 [12].

Supervised learning requires that each input vector be paired up with a target vector which represent the desired output. It is implemented by minimizing a measure of the difference between the discriminator output OUT and a corresponding target vector denoting the class of the training example. In back-propagation, the error measure is the mean-squared-error (MSE) objective function or cost function.

In order to find a set of weight parameters that will obtain a local minimum of the MSE objective function, an iterative search procedure, gradient descent, is used to successively improve the weight parameters from an arbitrary starting point by computing the gradient of the classifier's MSE with respect to the weight parameters using the chain rule.

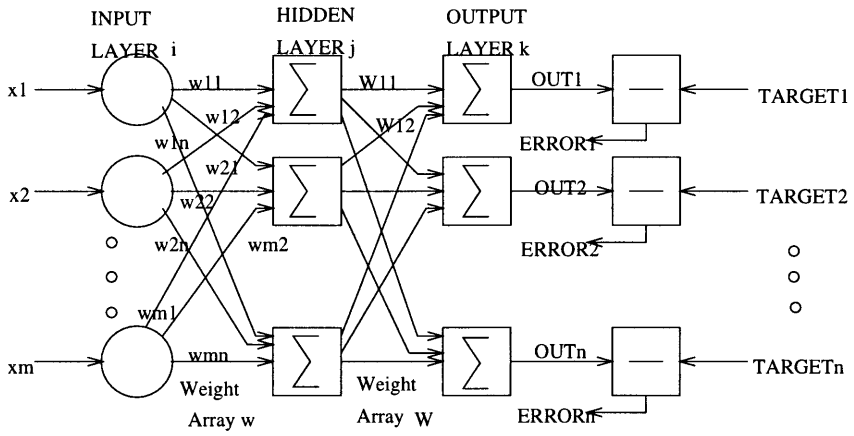

Figure 2-5: A Two-layer Back-propagation Network

Given that we know the weight parameters at the n^{th} iteration are W_n, then

$$W_{n+1} = W_n + \Delta W \tag{2.3}$$

The following is a step-by-step mathematical derivation to find ΔW for both the weights of the output layer and the hidden layer.

In a two-layer network such as that illustrated by Figure 2-5, notational conventions are as follows: input units are denoted by X_i, hidden units by H_j, output terminals by OUT_k, and desired outputs by D_k. There are connections w_{ij} from the inputs to the hidden units, and W_{jk} from the hidden layer to the output terminals. Please note that index i always refer to the input unit, j to a hidden one, and k to an output terminal.

Given that an input pattern from a set of training examples is applied to the

above two-layer neural network, then the output of the hidden layer neurons is

$$H_j = F(\sum_i w_{ij} X_i) \tag{2.4}$$

Thus, the final output of the network is

$$OUT_k = F(\sum_j W_{jk} H_j) = F(\sum_j W_{jk} F(\sum_i w_{ij} X_i)) \tag{2.5}$$

And the MSE objective function to be minimized with respect to the weight parameters is

$$MSE[w] = \frac{1}{2} \sum_n [D_k^n - OUT_k^n]^2 \tag{2.6}$$

where n is the number of input patterns in the training set. Substituting Equation 2.4 for OUT_k, Equation 2.5 becomes

$$MSE[w] = \frac{1}{2} \sum_n [D_k^n - \underbrace{F(\sum_j W_{jk} F(\overbrace{\sum_i w_{ij} X_i^n}^{h_j^n}))}_{L_k^n}]^2 \tag{2.7}$$

The objective function *MSE[w]*, which measures the system's performance error, has been written as a continuous differentiable function that only depends on the weight parameters w_{ij} and W_{jk} and input patterns X_i. According to the **gradient descent algorithm**, a change in the weight parameters, ΔW, is proportional to the gradient of *MSE* at the current position. Thus for the hidden-to-output connections the gradient descent rule gives

$$\begin{aligned} \Delta W_{jk} &= -\eta \frac{\partial \text{ MSE}}{\partial W_{jk}} && (2.8) \\ &= \eta \sum_n [D_k^n - OUT_k^n] F'(L_k^n) H_j^n \\ &= \eta \sum_n \delta_k^n H_j^n && (2.9) \end{aligned}$$

where δ_k^n is defined as

$$\delta_k^n = F'(L_k^n)\underbrace{[D_k^n - OUT_k^n]}_{ERROR_k} \tag{2.10}$$

For the input-to-hidden connections Δw_{ij} which are embedded deeper in Equation 2.6, the chain rule is used to compute the derivative:

$$\begin{aligned}
\Delta w_{ij} &= -\eta \frac{\partial \text{ MSE}}{\partial\ w_{ij}} && (2.11)\\
&= -\eta \sum_n \frac{\partial \text{ MSE}}{\partial\ H_j^n} \frac{\partial\ H_j^n}{\partial\ w_{ij}} \\
&= \eta \sum_n [D_k^n - OUT_k^n] F'(L_k^n) W_j k F'(h_j^n) X_i^n \\
&= \eta \sum_n \delta_k^n W_{jk} F'(h_j^n) X_i^n \\
&= \eta \sum_n \delta_j^n X_i^n && (2.12)
\end{aligned}$$

with

$$\delta_j^n = F'(h_j^n) \sum_k W_{jk} \delta_k^n \tag{2.13}$$

Note that Equation 2.12 has the same form as Equation 2.9, except with a different definition for the δs. In general, when updating the weight parameters connecting the layer i to layer j, the back-propagation updating rule always has the form

$$\Delta w_{ij} = \eta \sum_{patterns} \delta_{output(j)} * V_{input(i)} \tag{2.14}$$

where V stands for the appropriate inputs to layer i, which could be a hidden or real input layer. The definition of δ depends on the output layer. If it is the last layer of the network, it is given by Equation 2.9. Otherwise it is given by Equation 2.12 [5].

Chapter 3

Implementing Back-propagation in MATLAB

3.1 Reasons for Using MATLAB

The name MATLAB stands for matrix laboratory. It is an interactive system that provides fast matrix calculations. This is a very useful feature, since most of the numerical calculations in neural computing are matrix operations. MATLAB's excellent graphical features can also be utilized in examining error surfaces and in analyzing boundary decision diagrams. MATLAB also provides relatively easy-to-build graphical user interface. This can be used to construct real-time plotting and to allow user interaction during the training process.

However, whenever one programming language is chosen over others, there almost always will be trade-offs. Although, MATLAB is easy to program and it allows for a high degree of flexibility, compared to other low-level programming languages, such as C, it is considerably slower.

3.2 An Overview of Training

In the previous chapter, a mathematical derivation for updating the weight parameters was presented. The objective of training a neural network is to update the

weights so that the application of a set of inputs produces the desired outputs. The training process follows these steps:

1. Initialize all weight parameters to small random numbers (e.g., between 0 and 1).

2. Select an input pattern from the training set, and apply it to the network input.

3. Calculate the output of the network.

4. Calculate the error between the actual network output and the desired output (the target vector corresponding to the input pattern).

5. Calculate the gradient vector at the current position, and keep a running sum of all the gradient vectors results from applying a single input pattern from training set.

6. Normalize the accumulative gradient vector by its length, and calculate the appropriate weight updates.

7. Sum all the squared errors associated with each input pattern, and calculate the mean-squared-error (MSE) for the entire training set.

8. Check in which direction the MSE has changed, and adjust the learning rate to accelerate error convergence.

9. Repeat steps 2-8, until the MSE, or ΔMSE of the training set meets a preset tolerance

The nine steps itemized above can be illustrated more clearly by the system flow chart shown in Figure 3-1.

The entire training operation can also be divided into two major phases. One is known as the "forward pass" which constitutes steps 1, 2, and 3. This phase is very similar to the way in which the trained network will eventually be used to perform recognition tasks. Specifically, an input vector is applied, and an output is

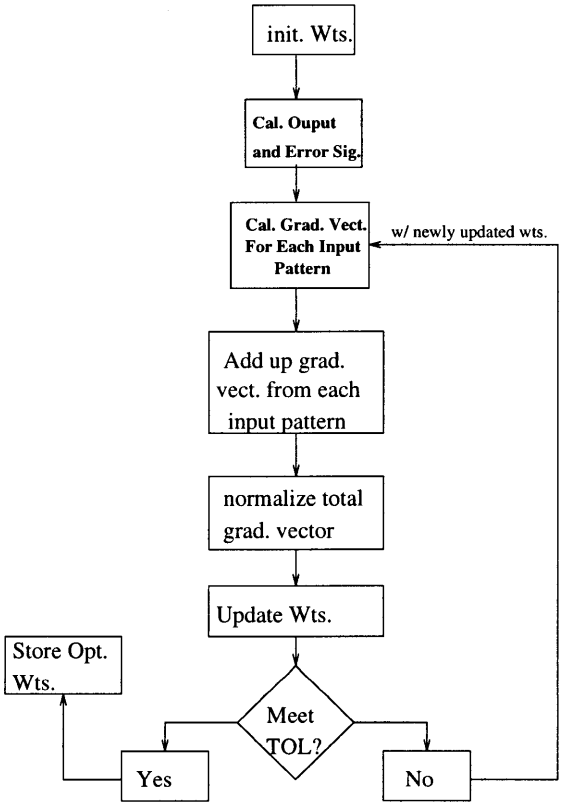

Figure 3-1: Neural Network Training System Flow Chart

calculated on a layer-by-layer basis. The second phase which is the "reverse pass" is the more interesting pass. The reason that it is called the "reverse pass" is that the error signal, the difference between the target vector and the actual output, is propagated backwards through the network layer-by-layer, where they are used to calculate the appropriate weight adjustments. The following sections will explain the implementation of the two passes in more detail [12].

3.3 Nerual Network System Flow Charts

The entire program is comprised of four major functions, each performing a distinct task. The overall structure of the program is illustrated in Figure 3-2.

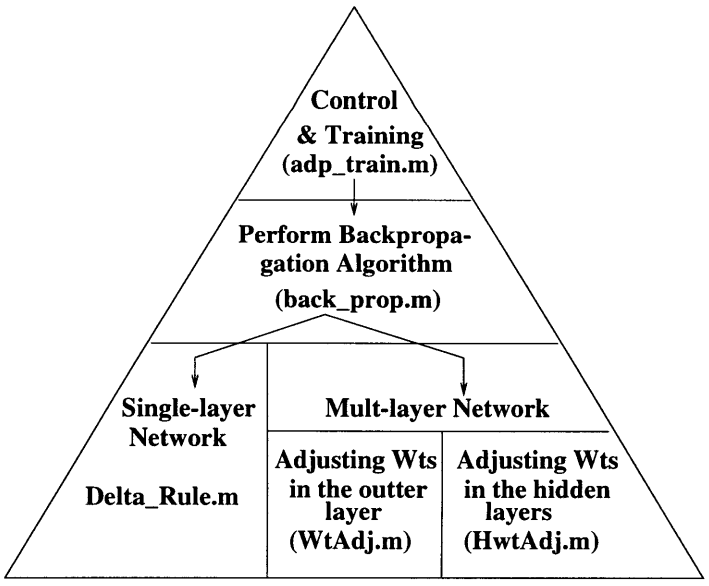

Figure 3-2: Neural Network Program Structure

3.3.1 Preparation Phase Prior To Training

Adptrain.m, is the file that runs the entire program. Initially, it will prompt the user to input relevant informations as outlined below:

1. Information about the input files.
   (a) Name of the training file.
   (b) Name of the corresponding target file.
2. Information about the network architecture.
   (a) Number of layers in the network (1 to 3 layers)
   (b) Number of neurons in each layer.
3. Stopping criterias for training.
   (a) Error tolerances.
   (b) Error goal.

This information is entered by the user prior to training. By including this information as input parameters to the program, it gives the user enough freedom to specify a particular training procedure for a specific problem at hand.

After taking in this information, the function will then proceed to load the necessary files required for training, computing their dimensions, and then moving on to the initialization phase. During the initialization phase, input data is normalized to a usable range, usually between -1 and 1. After normalization, the next step is to initialize parameters. The initial-weight-matrices are initialize randomly with numbers between 0 and 1. Other parameters include the MSE and weighted-delta-matrices; these should be initialized to zeros.

Another feature that is created during this time is the "DONE" button. A user can stop the training process at anytime by simply pushing the "DONE" button. Subsequently, the training process will stop, and all the updated weight matrices will be stored properly. This is a good feature to have, because a training process can

literally go on forever and never meet the stopping criterias; the "DONE" button allows the user to stop the training after a reasonable amount of time and save the training results thus far.

3.3.2 Training

When all the initial preparations have been completed, training will begin. First, the entire training set is trained once with one input pattern at a time. For each input pattern, the training result is calculated using the *backprop* function; this is the function that performs the actual back-propagation algorithm, which is in level two of the pyramid structure as shown is Figure 3-2.

Backprop takes the weight matrices from the previous training, unless it was the first training iteration, in which case it will take in the initial weight matrices instead. It will also take in one input pattern, which is usually a row vector in the training data set, and its corresponding target vector. After the training is completed, *backprop* will return the gradient vectors and the sum-squared-error.

First, *backprop* calculates the final output of the network, given its inputs and the weight matrices from each layer. For example, a two-layered network has a final output according to Equation 2.5:

$$OUT_k = F(\sum_j W_{jk} H_j) = F(\sum_j W_{jk} F(\sum_i w_{ij} X_i))$$

where, w_{ij} are connecting weights from the inputs to the hidden neurons, and W_{jk} are weight connections from the hidden neurons to the output units.

After outputs are calculated, *backprop* proceeds to perform the appropriate back-propagation algorithm depending on whether the network is a single or multi-layered. If the network is single-layered, the weight update algorithm used in this case is called the Delta Rule method. The basic equations used for calculating the Δw's are:

$$\Delta w_i = \eta \delta x_i w_i(n+1) = w_i(n) + \Delta w_i \tag{3.1}$$

where the term δ is the difference between the desired output D and the actual output A. In symbols,

$$\delta = (D - A) \tag{3.2}$$

and

$$\begin{aligned} \Delta w_i &= \text{the weight correction associated with the } i\text{th input } x_i \\ w_i(n+1) &= \text{the value of the weight } i \text{ after the adjustment} \\ w_i(n) &= \text{the value of the weight } i \text{ before the adjustment} \end{aligned}$$

However, instead of returning Δw_i each time, *Delta_Rule* function will return only the gradient vector, which is defined as the term $\partial MSE/\partial w$. So, it is obvious that,

$$\Delta w = -\eta * gradientvector \tag{3.3}$$

We return only the gradient vector after training each pattern in the training set because, after the entire training set is done training, we want to add up all the gradient vectors from each training iteration, and normalize them by their total length. This way each gradient vector will have a unit length, and when multiplied by η, which is also known as the *step size*, the weight correction factor will "step" into the direction of the gradient vector by a distance specified by η. *Delta_Rule* function also returns the squared error,δ^2, from each input pattern. These errors are also summed and then averaged (i.e., MSE) after the training process is completed.

Adjusting Weights in the Output Layer

The above describes the training procedure for a single-layered network. For a multi-layered network, the training task is slightly more complex, mainly because we have to adjust both the output and hidden weights. Adjusting the weights for the output layer is the easier of the two, because a target value is available for each neuron in the output layer. Recall Equations 2.8 and 2.9, together they devise the strategy for finding the weight adjustments for the output layer. First, these equations will be

simplified to show the training process for a single weight parameter connecting from hidden layer j to output layer k.

Recall Equation 2.10 defined a δ value,

$$\delta_k^n = F'(L_k^n)[D_k^n - OUT_k^n]$$

This equation can be simplified to just represent a single neuron r, in the output layer. Recall L_k^n was defined as the network output prior to the squashing function. The derivative of the squashing function (i.e., F') can be expressed in terms of the function itself (Equation 2.1). Thus, the δ value for a single neuron in the kth, or output, layer can be expressed as,

$$\delta_{r,k} = OUT_{n,k}(1 - OUT_{r,k})(Desired - OUT_{r,k}) \tag{3.4}$$

Then, according to Equation 2.9, where $\delta_{r,k}$ is multiplied by the output of neuron q, from the previous layer j, and then multiplied by the step size η, the result is the weight adjustment for the weight connection between neuron q in layer j, and neuron r in layer k. The same process is repeated for all the weight connections from a neuron in the hidden layer to a neuron in the output layer.

The following equations demonstrate the weight updating method for a weight connection from neuron q, in hidden layer j, to neuron r in output layer k:

$$\Delta w_{qr,k} = \eta \delta_{r,k} OUT_{q,j} \tag{3.5}$$

$$w_{qr,k}(n+1) = w_{qr,k}(n) + \Delta w_{qr,k} \tag{3.6}$$

$\Delta w_{qr,k}$ = the weight correction for the weight connection between neuron q in the hidden layer j and neuron r in the output layer k

$w_{qr,k}(n)$ = the value of the weight from neuron q in the hidden layer to neuron r in the output layer prior to the adjustment

$w_{qr,k}(n+1)$ = the value of the weight after the adjustment

$\delta_{r,k}$ = the value of δ for neuron r in output layer k

$OUT_{q,j}$ = the OUT value for neuron q in the hidden layer j

Figure 3-3 [12] shows the entire training process for a neuron r in the output layer k.

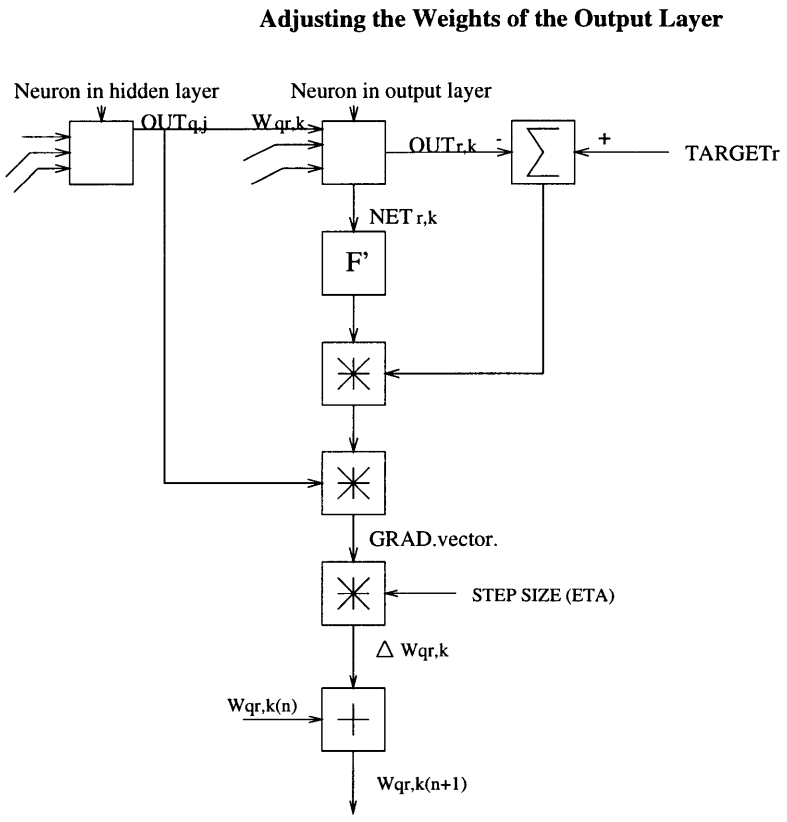

Figure 3-3: Training a Weight in the Output Layer

Adjusting the Weights of the Hidden Layers

Unlike the the output layer neurons, target vectors are not readily available for neurons in the hidden layers. Therefore the training procedure described above can not be applied here. However, recall Equation 2.11, which gives the input-to-hidden weight adjustments. These weights were embedded deeper in the objective function (i.e., as in Equation 2.5), thus the chain rule is required when we try to minimize the error function with respect to these hidden connections. From Equation 2.11, a

new training process can be deduced to train any weight connections that are not connected to the output layer [12].

Similar to the δ value defined in Equation 2.10, here we will also define a δ value, but with a slightly different definition. This was previously derived in Equation 2.13:

$$\delta_j^n = F'(h_j^n) \sum_k W_{jk} \delta_k^n \tag{3.7}$$

Once again, we reduce the equation above to represent a single neuron q, in hidden layer j. Recall h_j^n was defined as the hidden layer outputs; for simplicity we defined one of the outputs associate with neuron q in hidden layer j as $OUT_{q,j}$. The derivative of the squashing function with respect to $OUT_{q,j}$ (i.e., F') is $OUT_{q,j}(1-OUT_{q,j})$. W_{jk} was defined as the weight vector that included all the weight connections from each neuron in layer j to output layer k. But since we are only concerned with a single neuron q in layer j, W_{jk} is reduced to $w_{qr,k}$, which is defined as weight connections starting from neuron q, in layer j to all the neurons (a total r of them) in the output layer k, and each neuron in layer k has a δ value of $\delta_{r,k}$, and $\delta_{r,k}$ is computed using Equation 3.6. Thus for a single neuron q, in layer j, its δ value is,

$$\delta_{q,j} = OUT_{q,j}(1 - OUT_{q,j})(\sum_r \delta_{r,k} w_{qr,k}) \tag{3.8}$$

Notice that, in order to find the weight adjustments for the hidden layers, we must first compute the δ values and the updated weight connections with their destinations in the output layer. Then we will propagate the sum of all the products of $\delta_{r,k}$ and the updated $w_{qr,k}$, backwards in order to find the δ values for the hidden layer and subsequently the weight adjustments for the hidden layer. And hence the term "reverse phase" is used to describe this part of the training process [12].

After having computed $\delta_{q,j}$, we can calculate the weights $w_{pq,j}$, the value of the weight from neuron p in the input layer i to the neuron q in the hidden layer j, using Equations 3.6 and 3.7 changing the indices to indicate the correct layers.

The indentical computations are repeated for each neuron in the hidden layer, until all the weights associated with the hidden layer are adjusted. This process is repeated, moving back toward the inputs from the output layer weights, until all the

weights are adjusted. This procedure is shown in Figure 3-4 [12].

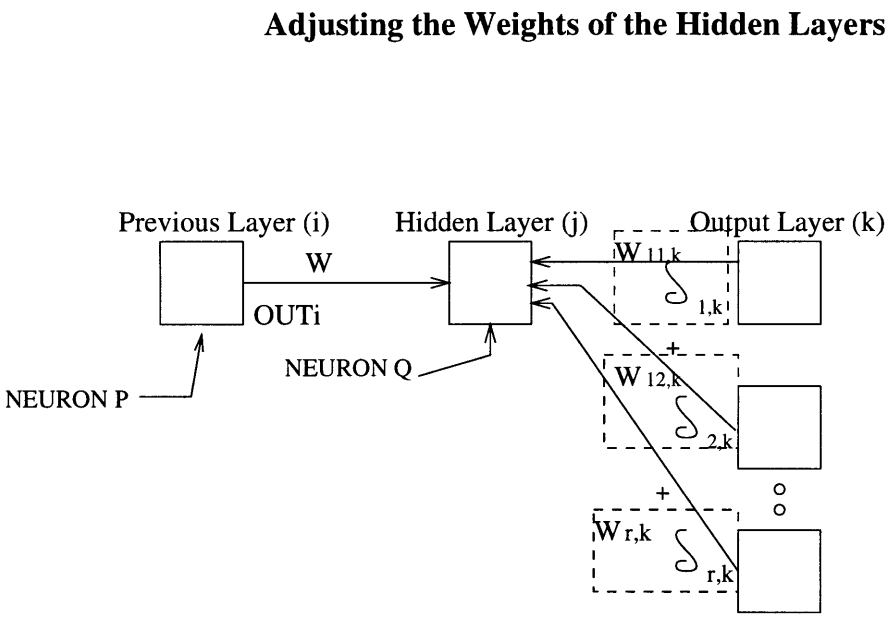

Figure 3-4: Training a Weight in the Hidden Layer

3.3.3 Modifications to Back-propagation

The basic back-propagation algorithm described above is exceedingly slow to converge in a multi-layer neural network, and many modifications have been studied to accelerate the convergence and improve its generalization ability. The following modifications have been implemented to quicken the training process.

Adding a Bias

One method which can possibly speed up the convergence of the training process is the use of a trainable bias added to each neuron. "This offsets the origin of the logistic function, producing an effect that is similar to adjusting the threshold of the perceptron neuron, thereby permitting more rapid convergence of the training process" [12]. In order to incorporate this feature into the training algorithm, we add

a weight connected to +1 to each neuron. This trainable weight is adjusted in the same way as all of the other weights, except that the source is always +1 instead of being the output of a neuron in the previous layer [12].

Momentum

Momentum is a method introduced by Rumelhart, Hinton, and Williams for improving the training time of the back-propagation algorithm. This method involves an additional term to the original weight adjustment that is proportional to the amount of the previous weight change. The modified weight update equations are now:

$$\Delta w_{qr,k}(n+1) = \eta(\delta_{r,k} OUT_{q,j} + \alpha[\Delta w_{pq,k}(n)]) \tag{3.9}$$

$$w_{qr,k}(n+1) = w_{qr,k} + \Delta w_{qr,k}(n+1) \tag{3.10}$$

where α is known as the momentum coefficient and is usually set to around 0.9 [12].

Adaptive Parameters

In the back-propagation algorithm, there are several constant parameters such as the training step size, η. The learning step size defines how much the weights should change in the direction of the gradient vector so to decrease the error function. If the step is too small, convergence will be very slow, if too big, instability is likely to occur [12].

One way to quicken convergence and at the same time preserve stability is to make the step size change adaptively to assist the training process. We would start the training process with a reasonably small step size proceed with the training process, and make the appropriate weight adjustments. After the adjustments have been made, we would then check to see if the cost function (i.e., MSE) had increased or decreased. If the cost function had been continuously decreasing it would be relatively safe to increase the step size by a small amount. But as soon as the cost function had started to increase, we should undo all the weight adjustments, and return to using the step size prior to the increase in the cost function had occurred. As to how the

step size should be increased, there are several methods. It is difficult to tell which method works the best other than by experimentation [11].

Chapter 4

Controlled Pattern Recognition Experiments

As an initial test of the neural network system, several controlled experiments were conducted to test its pattern recognition ability.

4.1 Multivariate Gaussian Classifier

This experiment is designed to test the neural network when exposed to a controlled data set. The input data set is a set of X and Y coordinates from several possibly distributions, each with a different mean and variance. Given the X and Y coordinates of a data point, the objective of the experiment is to correctly identify its corresponding distribution. Since the network has not yet been tested, initially we would also like to independently verify the performance of the network by traditional mathematical means.

4.1.1 Estimating Decision Boundaries

The following factors can make this classification process difficult: increased number of distributions, different a priori probabilities, $P(W_i)$, and arbitrary covariance matrices for each distribution. Therefore, in order to derive a relatively simple math-

ematical expression the following assumptions were made

1. Two Classes, W_1 and W_2.

2. Each Class is equally likely: $P(W_1) = P(W_2)$.

3. Each class has the same covariance structure: $\Sigma_1 = \Sigma_2 = \sigma^2 I$.

These assumptions transformed the process into a **minimum distance** classification problem. The minimum error decision rule is the squared **Mahalanobis** distance [18]. The discriminant function for distribution i is

$$G_i(X) = -\frac{1}{2}(X - \mu_i)^t \Sigma^{-1}(X - \mu_i) + logP(W_i) \tag{4.1}$$

where X is the input vector to the classifier, μ_i is the mean vector, Σ^{-1} is the inverse covariance structure, and $P(W_i)$ is the a priori probability.

From the discriminant questions we can find out the decision boundary between two classes by setting $G_1(X)$ equal to $G_2(X)$.

$$\begin{aligned} G_1(X) &= -\frac{1}{2}[\ x - \mu_{x,1} \quad y - \mu_{y,1}\ ] \begin{bmatrix} \frac{1}{\sigma^2} & 0 \\ 0 & \frac{1}{\sigma^2} \end{bmatrix} \begin{bmatrix} x - \mu_{x,1} \\ y - \mu_{y,1} \end{bmatrix} + logP(W_1) \quad (4.2) \\ &= -\frac{1}{2}\left[\frac{(x - \mu_{x,1})^2}{\sigma^2} + \frac{(y - \mu_{y,1})^2}{\sigma^2}\right] + logP(W_1) \quad (4.3) \end{aligned}$$

Similarly,

$$G_2(X) = -\frac{1}{2}\left[\frac{(x - \mu_{x,2})^2}{\sigma^2} + \frac{(y - \mu_{y,2})^2}{\sigma^2}\right] + logP(W_2) \tag{4.4}$$

Setting $G_1(X)$ equal to $G_2(X)$ and simplify the algebra, we have,

$$\begin{aligned} G_1(X) &= G_2(X) \quad (4.5) \\ (x - \mu_{x,1})^2 + (y - \mu_{y,1})^2 &= (x - \mu_{x,2})^2 + (y - \mu_{y,2})^2 \\ y &= \frac{\mu_{x,2} - \mu_{x,1}}{\mu_{y,1} - \mu_{y,2}}\mathbf{x} + \frac{\mu_{x,1}^2 - \mu_{x,2}^2 + \mu_{y,1}^2 - \mu_{y,2}^2}{2(\mu_{y,1} - \mu_{y,2})} \quad (4.6) \end{aligned}$$

4.1.2 Testing Neural Networks

Two Gaussian Distributions with Equal Variances

To test the neural network, the aforementioned problem is used. In this case, recall that both Gaussian distributions have the same covariance structure, and a priori probability. For class I, μ_1 is $(2, -2)$, for Class II, μ_2 is $(-2, 2)$. Substituting these values into Equation 4.6, it is immediately obvious that the classification boundary is a straight line $y = x$ as seen in Figure 4-1.

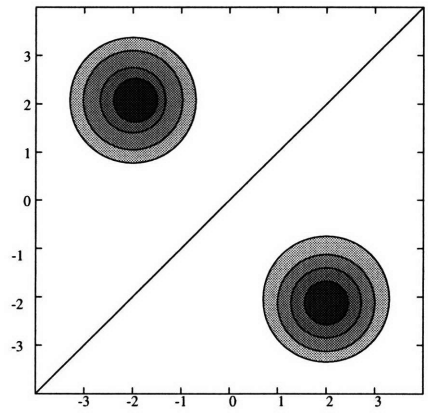

Figure 4-1: Two Gaussian Distributions with the Same Covariance Structures

After training the neural network for approximately 100 epochs, it produced results very close to those computed using the equations derived in section 4.1.1; the output of the network is shown in Figure 4-2.

As expected, the neural network made most of its classification errors near the decision boundary. A better illustration of these errors is shown in Figures 4-3 and 4-4.

Two Gaussian Distributions with Arbitrary Variances

Since the neural network successfully processed the simpler classification problem, we can now complicate the problem a bit further by varying the covariance structure for each distribution. In the example illustrated in Figure 4-5, Class I has a higher

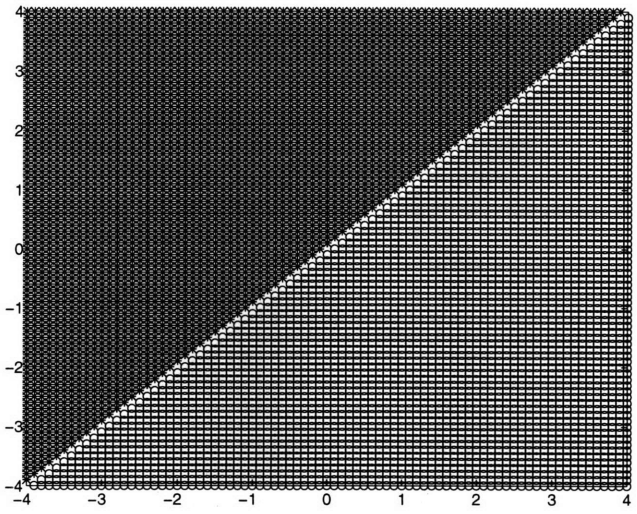

Figure 4-2: Neural Network's Classification Results on Two Gaussian Distributions with the Same Covariance Structures

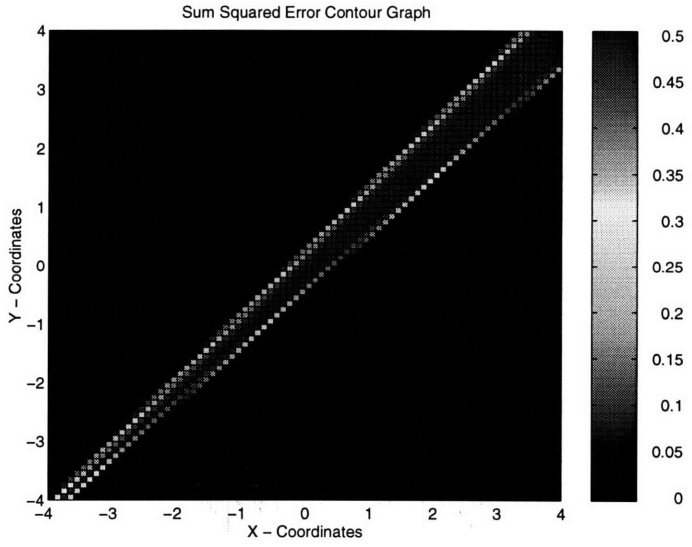

Figure 4-3: A Two Dimensional Error Contour Plot of the Neural Network's Classification Results

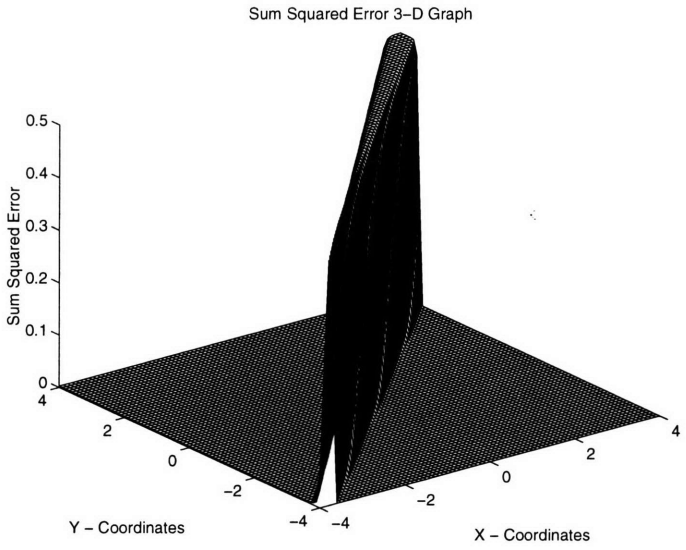

Figure 4-4: A Three Dimensional Mean-Squared-Error Plot of the Neural Network's Classification Results

variance in the y-dimension, so the distribution is elliptical. While Class II has equal variances in both x and y dimensions. In such cases, where the distributions have arbitrary convariance matrices, the decision regions are defined by a parabola.

After training the neural network for approximately 400 epochs, it obtained a parabolic-shaped decision boundary as we had expected. The output of the neural network is shown in Figure 4-6:

Again, as expected, the classification error is highest near the boundary region. Plots of the classification error is shown in Figures 4-7 and 4-8.

Four Gaussian Distributions with Arbitrary Variances

In the two previous tests, the outputs of the neural network was independently verified by mathematical computation. As more complicated classification conditions are presented to the neural network, it becomes increasingly difficult for us to verify its results through mathematical means. Instead we resort to intuition and human judgement to confirm the results of the neural network.

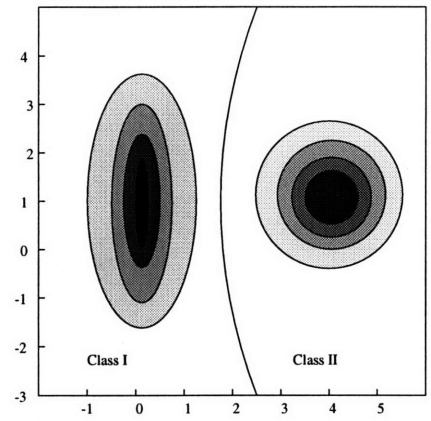

Figure 4-5: Two Gaussian Distributions With Arbitrary Covariance Structures

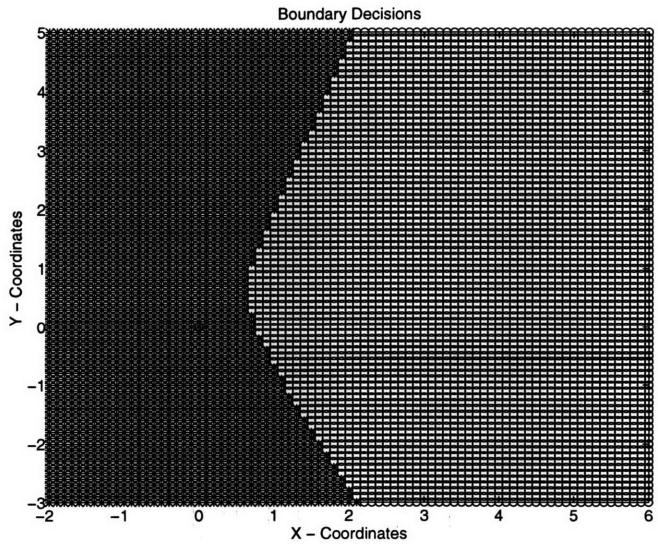

Figure 4-6: Neural Network's Classification Results on Two Gaussian Distributions with Arbitrary Covariance Structures

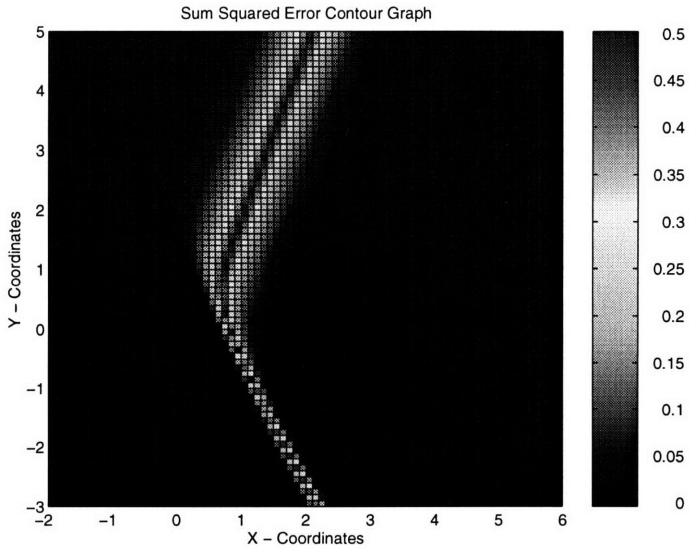

Figure 4-7: A Two Dimensional Error Contour Plot of the Neural Network's Classification Results

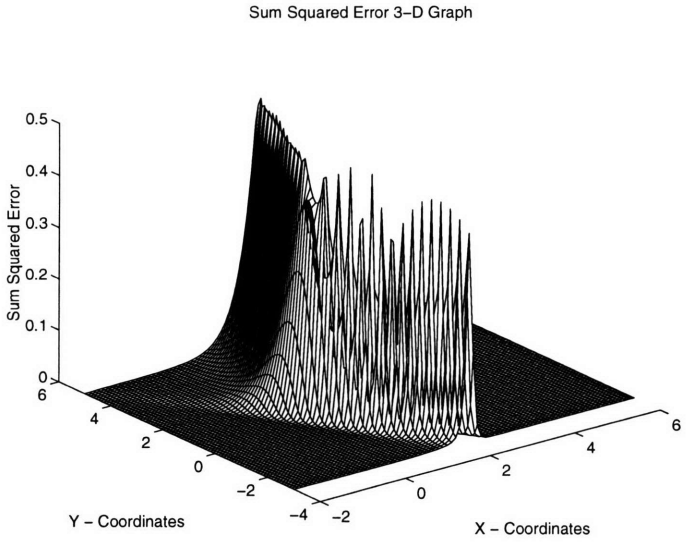

Figure 4-8: A Three Dimensional Mean-Squared-Error Plot of the Neural Network's Classification Results

In the following example, four Gaussian distributions each with a different covariance structure is presented to the neural network. These distributions are shown in Figures 4-9 and 4-10.

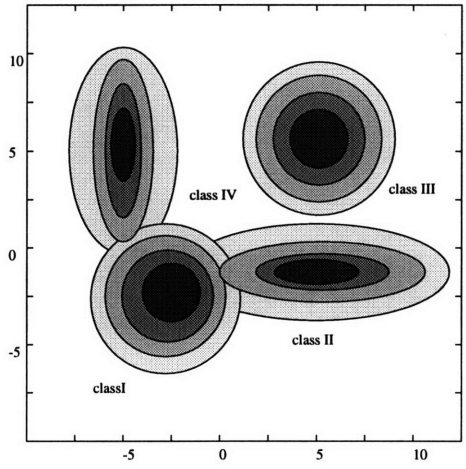

Figure 4-9: Four Gaussian Distributions with Arbitrary Covariance Structures

First, we constructed a training set consisting of 200 pairs of X and Y coordinates taken randomly from the four Gaussian distributions. After training the network for approximately 400 epochs, its mean-squared-error (MSE) reached 0.154, which is below the pre-set error tolerance. The output of the neural network is shown in Figure 4-11.

By inspection, the decisions boundaries are parabolicly shaped. They are consistent with our expectations; recall it was previously computed that in the case where distributions have arbitrary covariance matrices, the decision regions are defined a parabola.

Also, as expected most of the errors made are on or near the decision regions. In particular, the error is highest at the point where all four distributions meet. Figures 4-12 and 4-13 illustrate the classification errors.

From the results of these experiments, we can conclude that the neural network is capable of correctly processing classification problems.

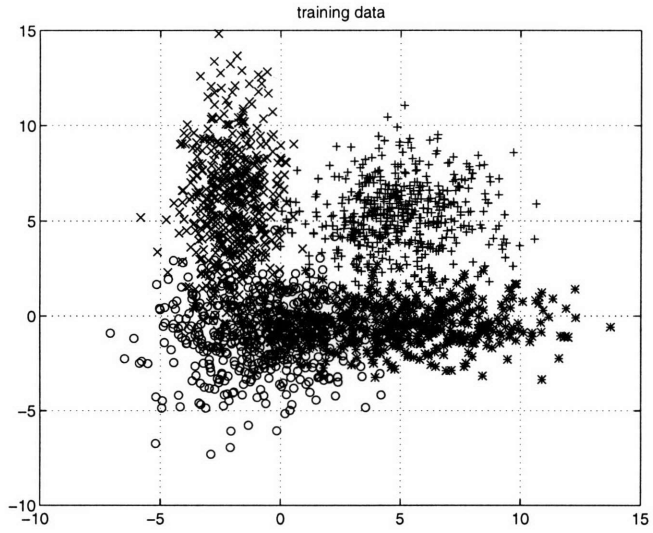

Figure 4-10: A MATLAB plot of Four Gaussian Distributions with Arbitrary Covariance Structures

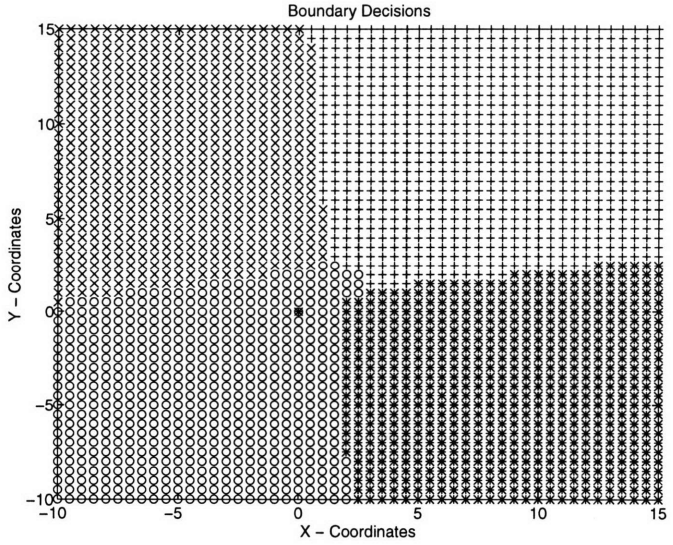

Figure 4-11: Neural Network's Classification Results on Four Gaussian Distributions with Arbitrary Covariance Structures

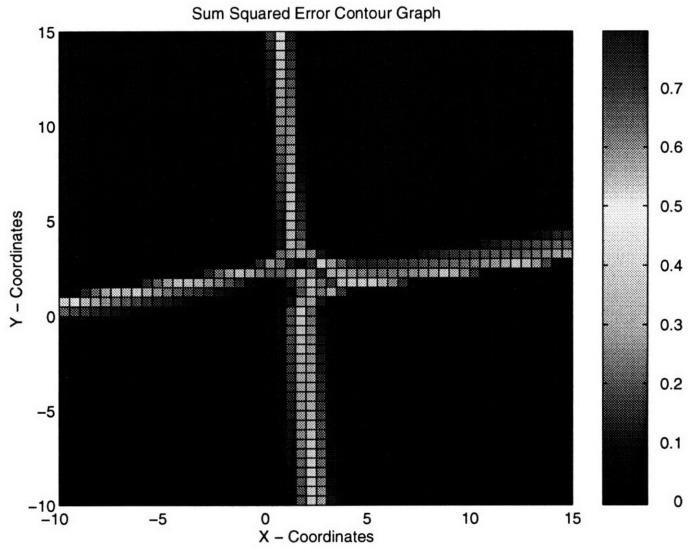

Figure 4-12: A Two Dimensional Error Contour Plot of the Neural Network's Classification Results

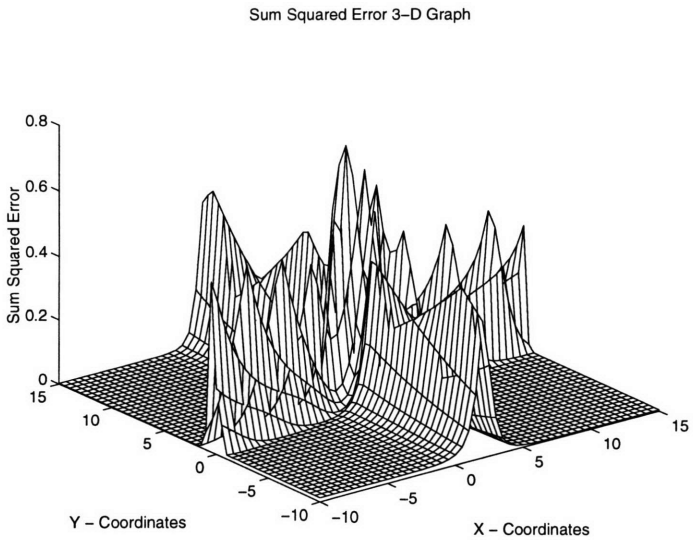

Figure 4-13: A Three Dimensional Mean-Squared-Error Plot of the Neural Network's Classification Results

4.2 Function Approximation Experiment

A neural network can also be used to estimate the functional relationship between the inputs and their corresponding outputs. In most applications, the function we are trying to estimate is either unknown or difficult to express in mathematical terms. In order to verify the neural network's ability to perform function approximation tasks, a known function, the square function, was used in the following experiment.

First, the input vector to the neural network had to be normalized to values between -1 and 1, because the output of an artificial neuron is bounded between -1 and 1. Then, the neural network was trained using a training set whose normalized values are $[-1, -0.99, -0.98 \ldots, 0.97, 0.98, 0.99, 1]$, and its corresponding target outputs were simply the values squared. This function is plotted in Figure 4-14.

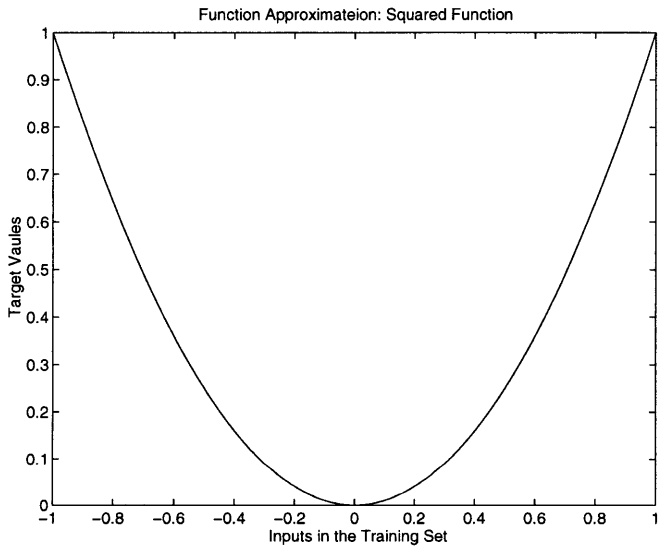

Figure 4-14: Training Set For the Neural Network

A series of error convergence plots is shown below. In Figure 4-15, the error seemed to be converging to a minimum, since it stayed almost completely constant from the 25th iteration to the 50th iteration. After the 50th iteration, we stopped the training process, and prepared a test set that contained data values different from the ones in

the training set. The actual test set used was $T = [-1, -0.975, -0.95 \ldots, 0.975, 1]$. The solid line in Figure 4-16 represents the desired output, and the dotted line represents the output of the neural network.

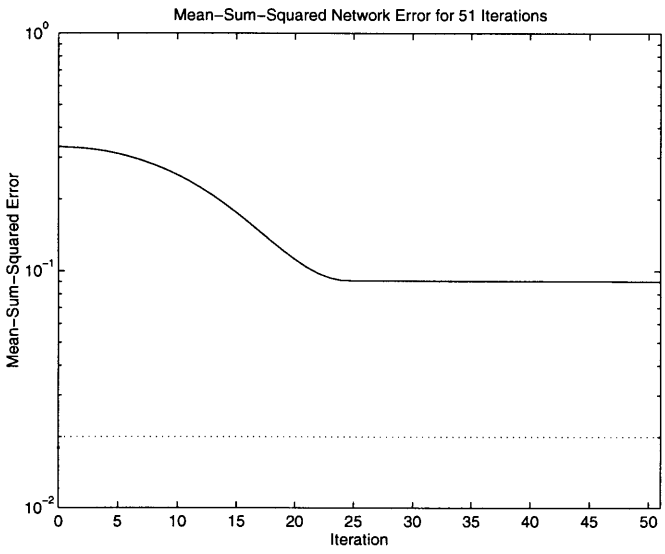

Figure 4-15: Error Convergence Plot After 50 Epochs

It is clear that the neural network's output is far from the desired output. To improve the network's performance, more training is needed. We ignored the "false" convergence seen in Figure 4-15, and continued to train the network. The final error convergence plot is shown in Figure 4-17.

Notice the aforementioned "false" error convergence is negligible. Soon after the 50th iteration, the error decreased significantly. Then, near the 160th iteration it began to level off slowly and eventually approaching a true minimum at the end of the training process. In the following figures, the outputs of the neural network are shown sequentially after the completion of the 100th, 150th, 200th and finally the 600th iterations.

The series of plots shown above is a good demonstration of the neural network's learning process. After the 50th iteration the neural network was only able to generate a linear function as shown in Figure 4-16. Then after the 100th iteration, the neural

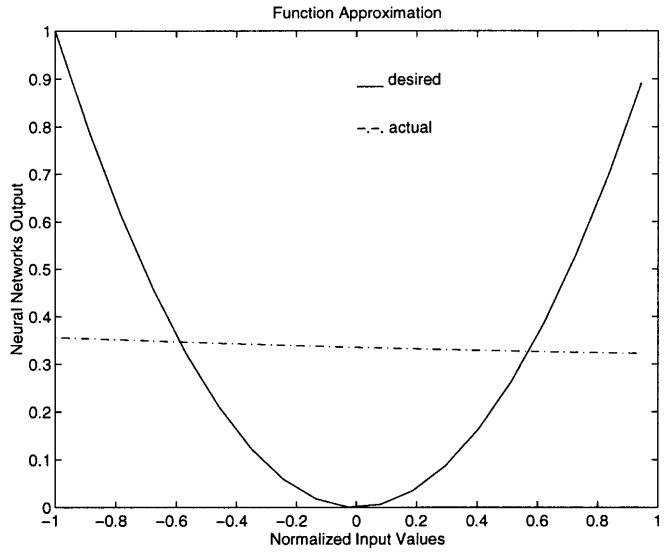

Figure 4-16: Neural Network Simulation Results After 50 Epochs

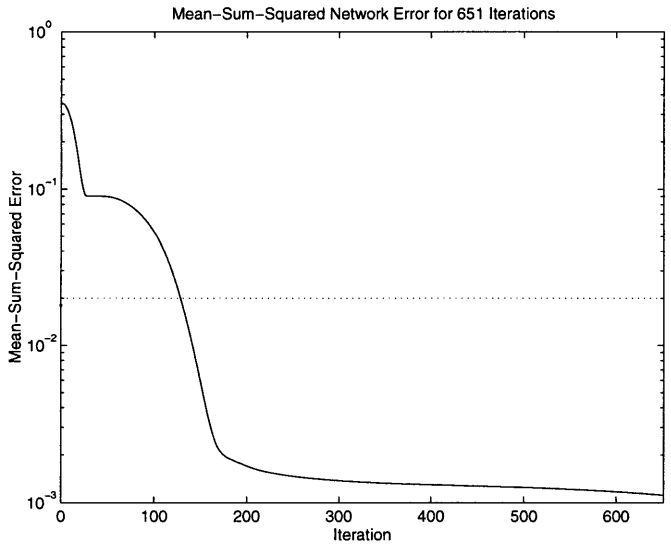

Figure 4-17: Neural Network Training Error Convergence Plot

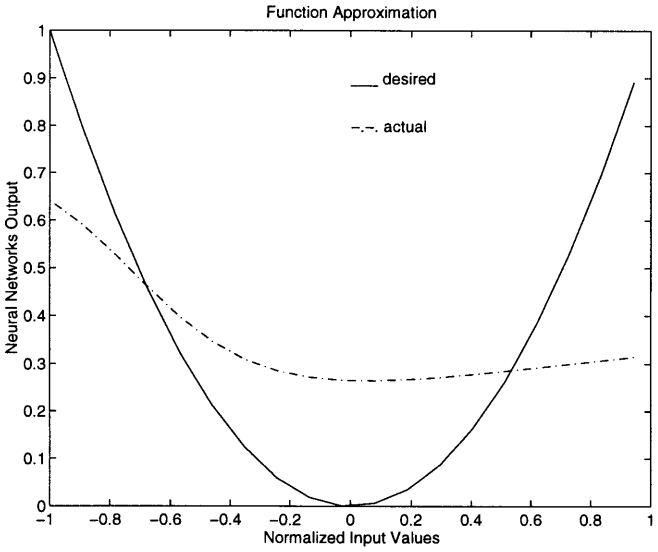

Figure 4-18: Neural Network Simulation Results After 100 Iterations

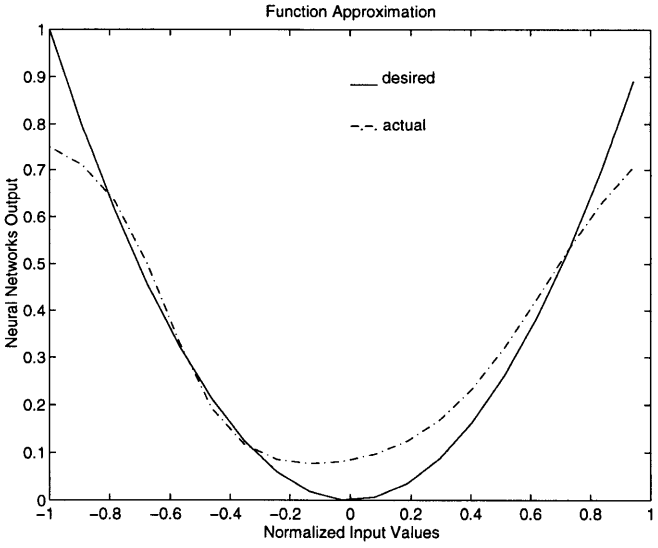

Figure 4-19: Neural Network Simulation Results After 150 Iterations

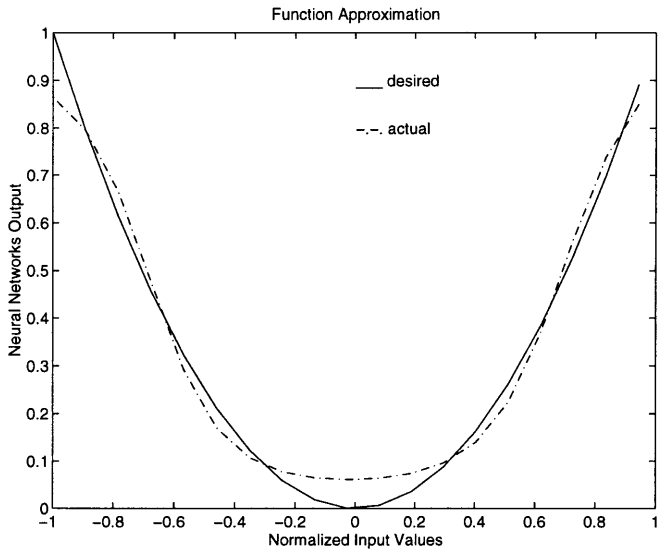

Figure 4-20: Neural Network Simulation Results After 200 Iterations

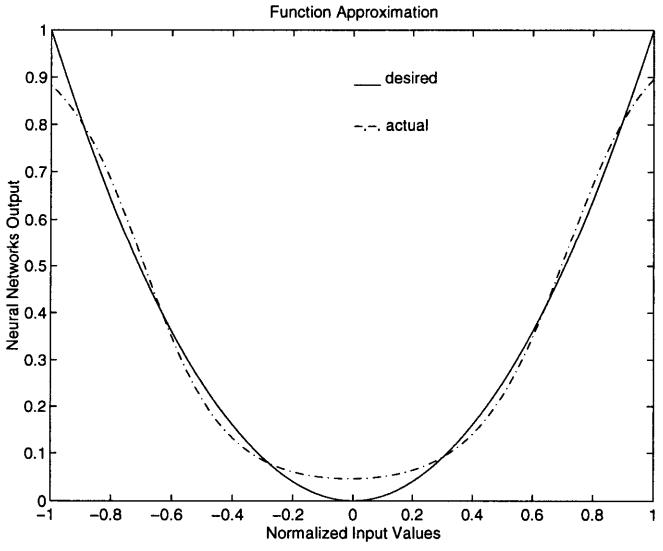

Figure 4-21: Neural Network Simulation Results After 600 Iterations

network started to learn the nonlinear relationship between its input vector and its corresponding target vector. In between the 100th and 200th iterations, the network has completed most of its learning. Hence, after the 200th iteration the output of the network was very close to the actual parabolic curve it is trying to estimate. After the 200th iteration, the error in the network was decreasing moderately implied that not much more learning was being done. Thus the final output of the neural network, which was taken after approximately the 600th iteration, is only slightly better than the result taken after the 200th iteration as shown in Figure 4-20.

These test results give us reasons to believe that the neural network is indeed working properly. In the next chapter, the neural network will be used to perform fault detection using real data collected from Ka-Band Link Experiment.

Chapter 5

Fault Detection Using Data From Ka-Band Links Experiment

In the previous chapter, a series of tests were conducted to test the reliability of the neural network software. After a sufficient amount of testing had been done to verify that the neural network software is working properly, it can be used to analyze real data collected from the antenna's downlink system, Downlink Analyzer.

5.1 Data Used for Fault Detection

The data used for the following experiments were collected from the Mars Observer Ka-band Link Experiment (KaBLE) [9]. A brief description of the KaBLE experiment is given, and the reasons for using the KaBLE data will also be explained.

5.1.1 Mars Observer Ka-Band Links Experiment

The Ka-Band Link Experiment is the first demonstration of a deep-space communications link in the 32 to 35GHz band (Ka-Band). It was designed to investigate the performance benefits of a shift from X-band (8.4 GHz) to Ka-band(32 GHz). It was carried out using the Mars Observer spacecraft while the spacecraft was in the cruise phase of its mission and used a 34-meter beam-waveguide research and development

antenna, DSS 13, at the Goldstone complex of the DSN [9].

This experiment has been going on for more than a year now, therefore there are plenty of data, especially good data, available for our study. Usually, at the early stage of an experiment, data collected are very unreliable because the equipment is still being adjusted to track the satellites. In addition to being abundant and reliable, the KaBLE data can be partitioned into a single *principal health metric, h*, and a set of *causal factors,* c_i. The following section will explain how the data is separated into the right format, so it can be used as an input to the neural network for fault detection. One last reason for using the KaBLE data is that all the c_i data collected are sufficiently low-level measurements so they are ideal both for studying the data relationships and later on for fault diagnosis, where the aim is to isolate the lowest-level cause(s) of an anomaly[15].

5.1.2 Partition of the KaBLE Data

The Downlink Analyzer is a tool for performing model-based fault detection, and diagnosis using a non-parametric neural network model. The diagnostic module will eventually isolate a detected fault to one or more of the input causal factors. These causal factors are raw time series data that we have carefully selected and obtained from KaBLE. They were mainly comprised of streams of monitor data from the Monitor and Control subsystem, and data from both the Antenna subsystem and the Electronic Tone Tracker (ETT).

Figure 5-1 shows the names of the data we have selected to use as the input to the neural network model, and the *Principal Health Metric, Pc/No*, which is used to determine the empirical health state of the downlink system.

Principal Health Metric

The *principal health metric, Pc/No* is selected such that a simple thresholding function, $\Omega_h(h)$, can be formulated to generate the empirical health state of the system. The principle health metric, h, is a stochastic process subject to underlying random

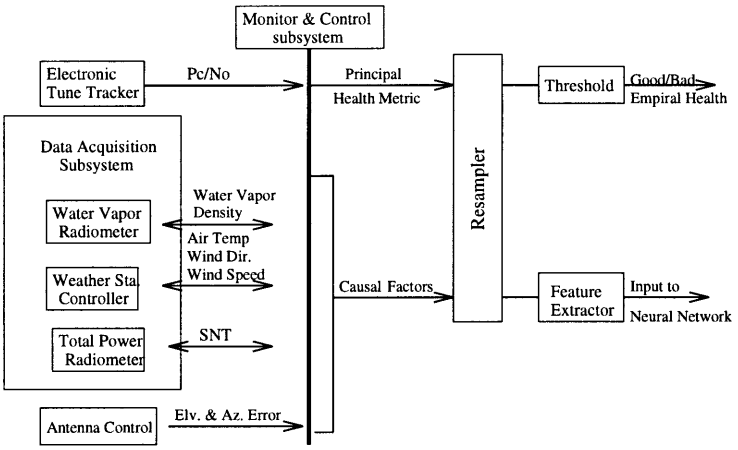

Figure 5-1: Input Data to the Neural Network Model

process noise, measurement noise, and time delays; but since it is frequently updated and estimated with high statistical confidence, in general we can take the empirically derived Ω_h to be equal to the actual system health state, Ω [15].

Causal Factors

The elements of c_i (i.e., the causal factors) are selected such that an arbitrary complex function $\Omega_c(c)$ can be constructed that also yields the correct system health state, $\Omega_c = \Omega$, without having to examine the principal health metric, h [15].

Figure 5-2 demonstrates that the function Ω_c, which exists among the causal factors, is very complex. We can see that the function that can eventually transform these causal factors into the principal health metric is extremely difficult to express in mathematical terms. That is because this function is complex and too difficult to be modeled mathematically. Neural networks extract the relationship that exists between the inputs without using physical modeling. However, having a general understanding of the overall model is helpful in network optimization.

The Wind and Antenna model is extremely complicated. With this model, a

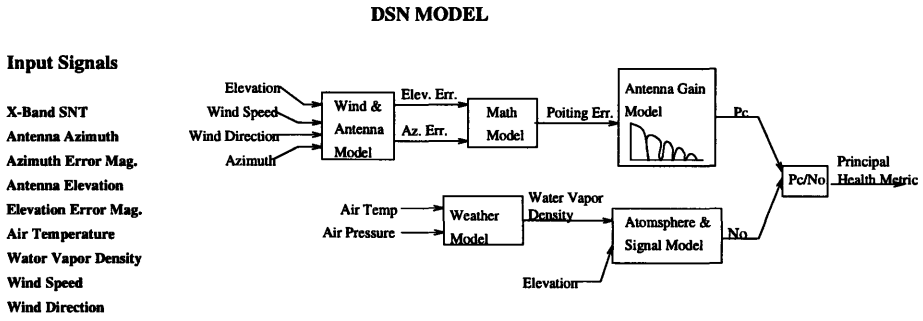

Figure 5-2: DSN Model to Produce Empirical Health State from the Causal Factors

knowledge of the antenna position (its elevation and azimuth angles), and the wind speed and direction, should make it possible to predict the azimuth and elevation errors. Currently, at JPL several attemps have been made to model the effect of wind on the antenna, such as the Finite Element Model(FEM). This is difficult to do, because the antenna has many mobile components, and there are numerous states needed to model the coupling between the different modes [3]. From the elevation and azimuth errors we can compute the pointing error (derivations shown later); and knowing the pointing error of the antenna, and its antenna gain patterns, we are able to calculate the amount of power received by the antenna, which corresponds to Pc, the numerator of the principal health metric.

The System Noise Temperature(SNT) is closely related to the atmosphere conditions surrounding the antenna. The noise level in the system changes depending on whether it is a clear or foggy day, and on the amount of atmospherical obstruction the antenna encounters. Also the position of the antenna is important; in particular, the antenna elevation will affect the amount of atmosphere obstruction the antenna is exposed to. And the azimuth position will determine the antenna's position in relation to the sun. As we know, radiation from the sun also affects system's noise

level. Thus, atmopsheric information such as water vapor density and antenna position help to determine the System Noise Temperature, *No*, which is the denominator of the principal health metric.

The function Ω_c can be constructed to reproduce the principal health metric value, *Pc/No*. The process of constructing the DSN model helped to identify the inputs to be included in the input data set. The neural network model will also provide information about the relationships among the causal factors, and recognize patterns in the input data in relation to the health state of the system that can not be easily computed by other methods.

5.2 Using the Neural Network Model to Analyze KaBLE Data

As it was demonstrated earlier that it is extremely difficult to obtain a suitable parametric model for the downlink system and to determine an appropriate model order from empirical data. However, a neural network, connectionistic model implements a direct mapping from the input elements to an arbitrary complex function Ω_c, which can produce the correct health state, Ω without examining the principal health metric, h [15].

5.2.1 Selecting a training set

A training set is constructed from the original data consisting of feature vectors and empirical health state pairs selected to be representative of the underlying input space. Ideally, the training set provides uniform coverage of the multidimensional input space such that if the network successfully learns those patterns, it will exhibit good performance when confronted with unseen feature vectors during its normal operating regime [6].

Prior to constructing the training set data were gathered and examined carefully. The training set consisted of data from all the different collection times. Approxi-

mately half of the data is associated with a "bad" system health state and the other half is associated with a "good" system health state. Within each of the "good" and "bad" data segments each input vector is examined individually to see how their values cluster as a function of the health state of the system.

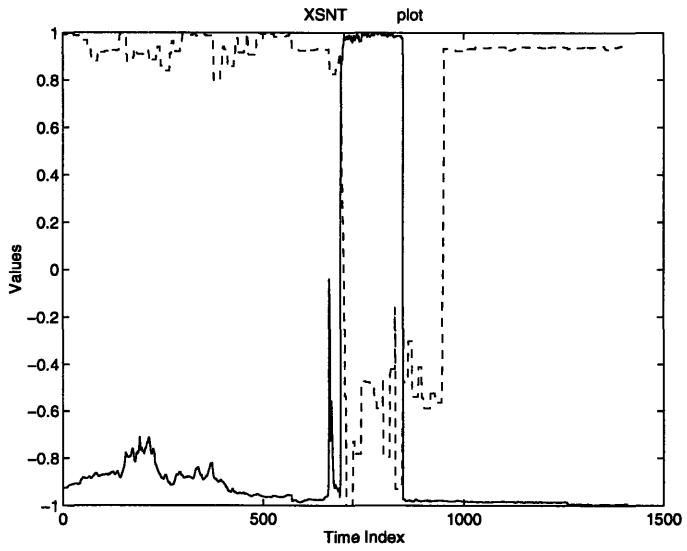

Figure 5-3: SNT and the System Health State

In Figure 5-3, the dotted line represents the health state of the system, (i.e, the values of Pc/No), and the solid line represents the SNT values. These signals are normalized to values between -1 and 1. We observe that there is an inverse relationship between the SNT values and the health state of the system; and the SNT values can be divided into two classes. One class of values are associated with the "good" health states of the system, the other is associated with the "bad" health states. This separation is shown in Figure 5-4; where, we definitely see a clear clustering of SNT values. This clustering effect is exaggerated in Figure 5-5.

In the training set, the SNT input vector has to be comprised of values that uniformly cover the "GOOD" range and the "BAD" range. Also, it needs to include values that are scattered in the middle, so the network can be trained to interpolate these values to the appropriate class. However for some of these ambiguous values

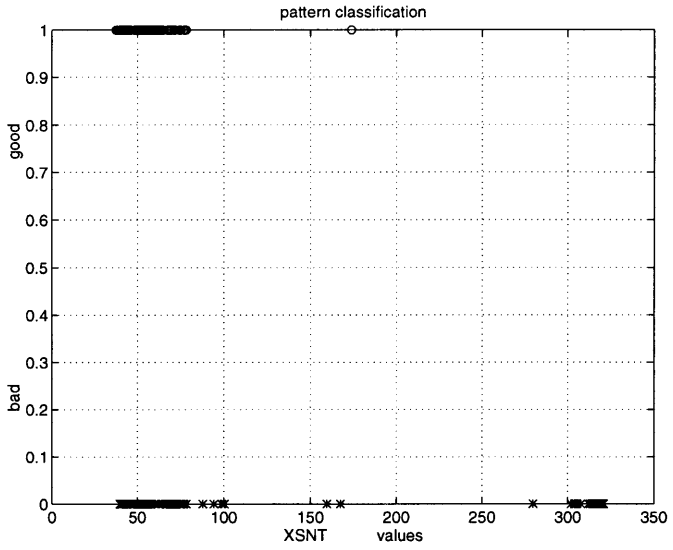

Figure 5-4: SNT Classified as a Function of the System Health State

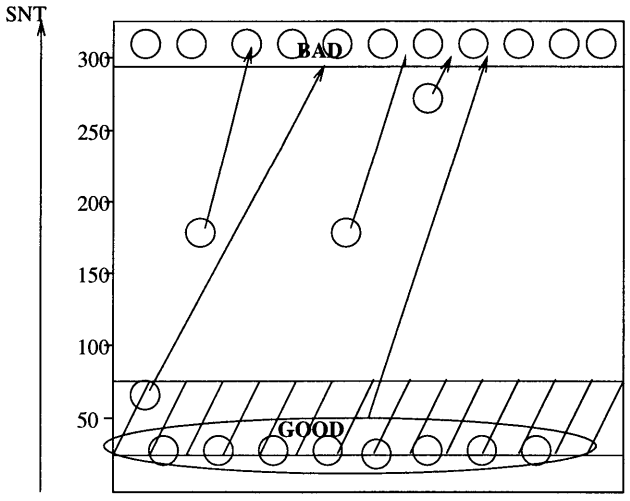

Figure 5-5: SNT Clustering

(i.e., the "BAD" circles in the "GOOD" range) it is almost impossible for the network to separate, and that is a big source of error.

For the remaining input vectors the same procedure was followed in order to attain good coverage which will help the network to best generalize on any unseen data.

5.2.2 Analysis Tool

In order to analyze the data as described in the previous section efficiently, the menu-driven graphical analysis tool shown in the Figure 5-6 was created.

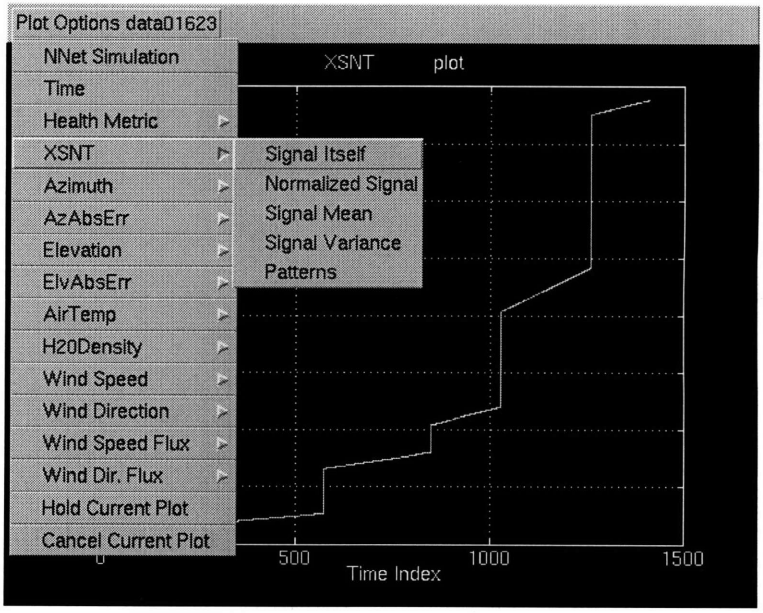

Figure 5-6: Menu for the Analysis Tool

Inside MATLAB, one can start the menu by typing:

>> plot_menu filename

where filename is the name of the input data file. Each data file contains a number of input vectors. A list of their names is shown when the "Plot Options Filename" button is pressed. In addition, a series of five options is applied to each input vector. These options allow one to look at the original signal or the normalized signal. Also

one can see how the input vector's values are separated into "Good" and "Bad" classes according to the system's principle health metric. An example of this was shown in Figure 5-4.

Sometimes it is useful to examine some of the basic statistical characteristics of the input vectors. This analysis tool is equipped with capabilities to calculate the mean and variance of a signal. These statistics are computed by first applying a moving window of fixed size, and passing it along the input vector. Inside the window the mean and variance of the signal samples are computed and recorded. For example, Figure 5-3 shows the SNT input vector, and we expect the mean SNT will be a smoothed version of the original signal, which is shown in Figure 5-7.

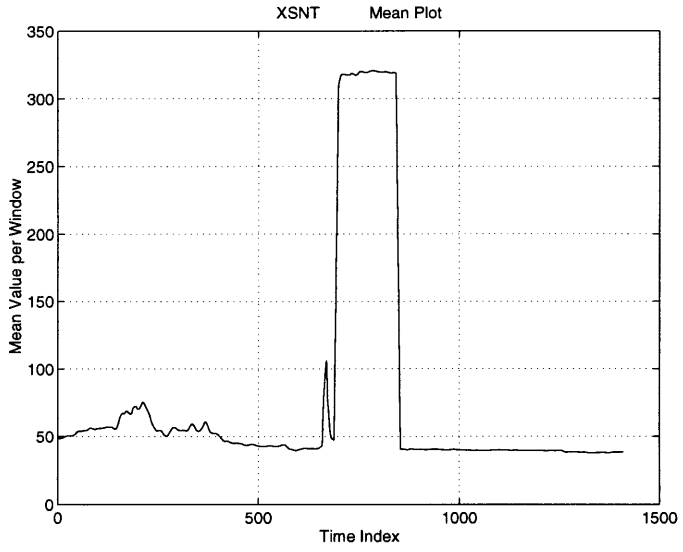

Figure 5-7: Mean SNT

The variance of the signal is shown in Figure 5-8, where the two big spikes correspond to the sharp changes in value. This is a very useful tool, often can be used to study input vectors. It is also helpful in identifying the effective input vectors to be included in the input data file.

Another feature in this tool set, is the "NNet Simulation" option. Once the network has been trained, this will simulate the detection results of the neural network

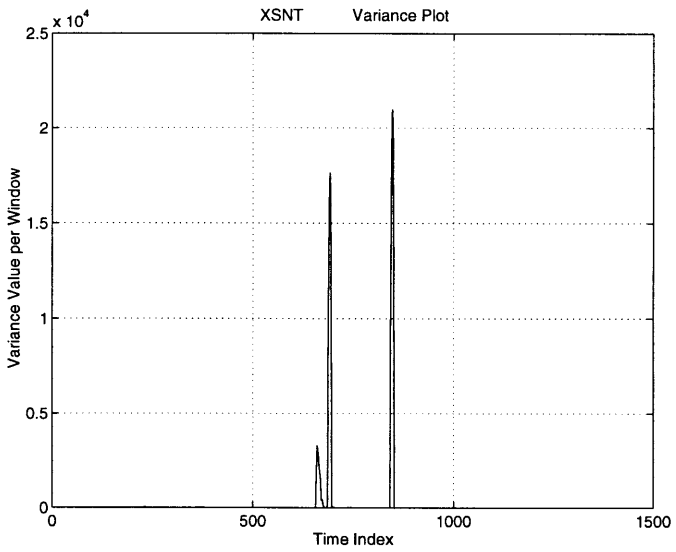

Figure 5-8: SNT Variance

using the test data.

In Figure 5-9, the dotted line is the principle health metric, *Pc/No*. A simple thresholding function was formulated and was applied to the principle health metric, and yield the empirical health state of the system. A "good" health state is represented by the value "1" on the vertical axis, while the value "0" represents a "bad" health state. These values are plotted using small circles.

Then the neural network was used to detect the health state of the system. Outputs of the neural network are also shown in Figure 5-9. They are plotted immediately below the empirical, or actual, health state of the system. Small "*"s are used to represent the "good" and "bad" health states determined by the neural network. Notice, near the 500 time index, the actual health states of the system were bad; but the neural network classified them as good.

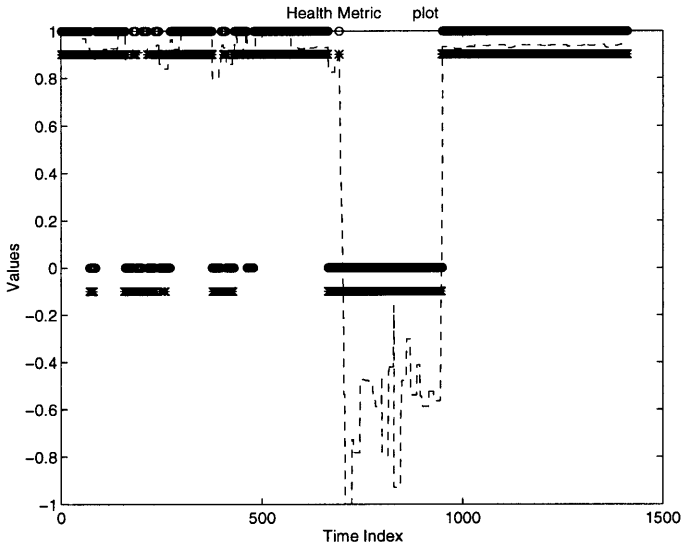

Figure 5-9: Neural Network Simulation Results

5.3 Methods to Optimize the Neural Network

Careful study of the data demonstrated that many optimization techniques can be made to increase the fault detection accuracy and to shorten the training time. The following experiments are some of the optimization techniques used with their results.

5.3.1 Feature Extraction

Feature selection is a procedure whose function is to extract from available data those features that appear most helpful for classification purposes. By selecting the appropriate features we can remove away some of the complexity from the neural network model [10].

However, prior to making any feature extractions the neural network was trained with the original raw data we obtained from the KaBLE experiment. There are a total of eight input signals:

1. System Noise Temperature (SNT)

2. Antenna Azimuth Angle

3. Azimuth Error

4. Antenna Elevation Angle

5. Wind Speed

6. Wind Direction

7. Air Temperature

8. Water Vapor Density

In this training procedure, a two-layer network architecture that has four hidden nodes and two output nodes was selected. This network's basic architecture is shown in Figure 5-10. The original data set had an enormous range of numbers because the values of the input vectors vary greatly. Thus the first step was to normalize each input vector to numbers between 1 and -1. After the training was completed, the result was stored in the two output layer neurons. The top neuron represented the "Good" health state, while the lower neuron represented the "Bad" state. Each neuron could produce an output value between zero and one. The one with the higher output was interpreted as designating the system's health state determined by the neural network. The higher the differential, δ, between the two neurons' output values the more confident we are about the network's detection result. Results from this initial training process is recorded in Table 5-1.

The training set was first trained until a pre-set mean-squared-error (MSE) criterion had been met. The neural network was able to achieve a 92.3% accuracy on the training set with a MSE value of 0.118. When tested with the test data set, its detection accuracy went down to 65.2%. Many reasons could account for this significant decrease in accuracy rate. It could be that the network wasn't complex enough to learn all the possible patterns. Or there might not have been enough information provided in the training set. Also, it could be that the training set wasn't a good

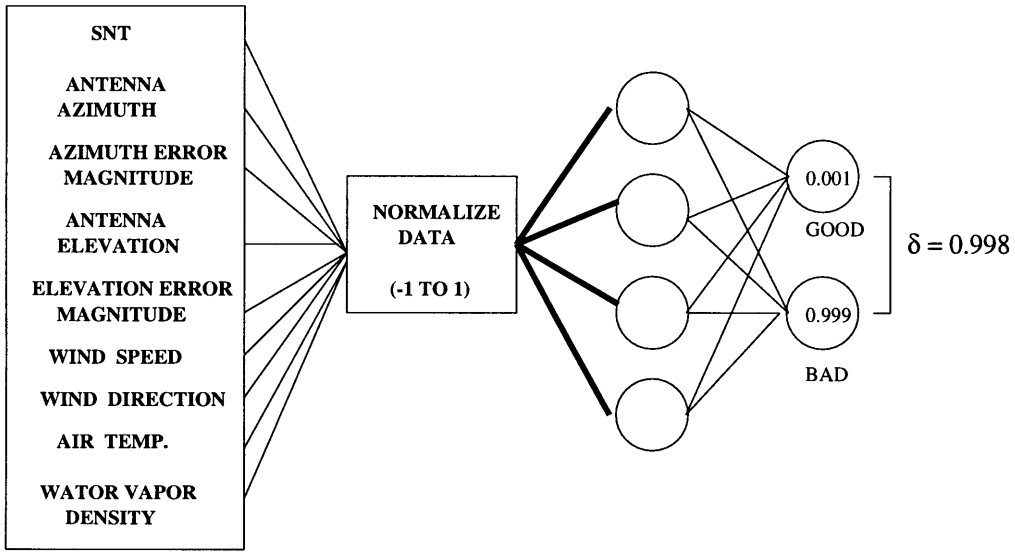

Figure 5-10: Input Data to the Neural Network Model

Table 5-1

Results from Testing with Original Raw Data

Stats	Training set	Test set
Accuracy	92.3%	65.2%
MSE	0.118	0.642
Type I Error	2.5%	31.3%
Type II Error	5.7%	3.5%

representative of all the data, thus the network can not generalize successfully on unseen data.

The other two remaining categories are Type I, and Type II errors. Type I errors are also known false alarms, meaning the actual health state is good, but the neural network has classified it as bad. A Type II error is also called a miss, meaning the actual state is bad, but the neural network has classified it as good. In our case, a lower Type II error is preferred, because it is more important that the downlink analyzer is able to detect all the faults in the system with the highest possible accuracy.

Adding Inputs

After the neural network was trained on the training set, it was then tested itwith a new set of data labeled as the test set, which included data that was not in the training set. It is clear that the neural network was not able to perform as well on the test set, as it did on the training set. Several methods are available to improve the network's generalization ability, hence increase its detection accuracy on the test set. One method is to modify the original input data set, which only included raw time series data obtained from the experiment, by either adding useful or deleting unimportant inputs.

One important question is what factors are most likely have caused the health state of the antenna to go bad? It was discovered that there is a strong correlation between the wind gust and the antenna's pointing error. In the original input data we only have information about the wind's direction and speed. To detect a wind gust, we can measure how much and how fast both the wind's speed and direction are changing. This can be accomplished by using the **Feature Vector Extraction** tools of the DLA Front End Processing. First, a **.feature** file was created; inside which names of the desired features were specified. In this case we calculated the variances for both the wind's speed and its direction. Variances of these signals describe the deviations from their means, which is what wind gust is defined as.

Adding Wind Speed Variance and Wind Direction Variance to the input data set can potentially complicate the neural network. However, if these additional inputs are

useful for fault detection purpose, they will be able to remove some of the complexities from the neural network and transfer them into the input. For example, a single layered network is only capable of modeling linear separable functions. In that case, if one needs to approximate a function such as $y = x^2 + x$, which is clearly not linear in x, then merely having x as the input, will not enable a single layered network to approximate the nonlinear function. However, if we have both x and x^2 as inputs, then this becomes a linear separable problem, and can be solved using a single layer network. In essence, by putting the extra useful input, x^2, into the input data set, we have effectively reduced the complexity of the problem to be approximated by the neural network, hence improving the network's pattern r

ecogni tion performance. Of course in our case, the situation is not as simple, because we don't know exactly what function the neural network needs to approximate, and thus we don't know what are exactly the most useful inputs.

After adding the two new inputs, Wind Speed Variance and Wind Direction Variance, into the input data set, the neural network was re-trained under the same exact conditions as the previous test. Results from this test is summarized in Table 5-2.

Table 5-2
Results from Testing with Wind Variances

Stats	Training set	Test set
Accuracy	95.6%	85.7%
MSE	0.079	0.268
Type I Error	1.5%	10.1%
Type II Error	2.9%	4.2%

These two additional inputs made a significant difference in the network's performance. The percentage of detection accuracy has increased from 65.2 to 85.7 percent. It is clear that Wind Speed Variance and Wind Direction Variance are important feature vectors that are critical for detecting faults and should be included in the input

data set.

Reducing the Number of Inputs

After more careful examinations of the original input data set, we discovered another method that could possibly reduce the network's complexity. We constructed a general DSN model, which is shown in Figure 5-2. One component of the DSN model is the Math Model that is used to calculate the antenna's pointing error. It is reasonable to believe that the pointing error is most directly related to Pc, power received by the antenna, than it is for the elevation and azimuth errors. This realization led to the replacement of both the Elevation Error and Azimuth Error inputs with a single Pointing Error. The pointing error is a function of the known elevation and azimuth errors. The mathematical relationship $P.E. = \mathcal{F}(E.E., A.E.)$ is found found using a geometric approach, which is depicted in the diagram shown in Figure 5-11.

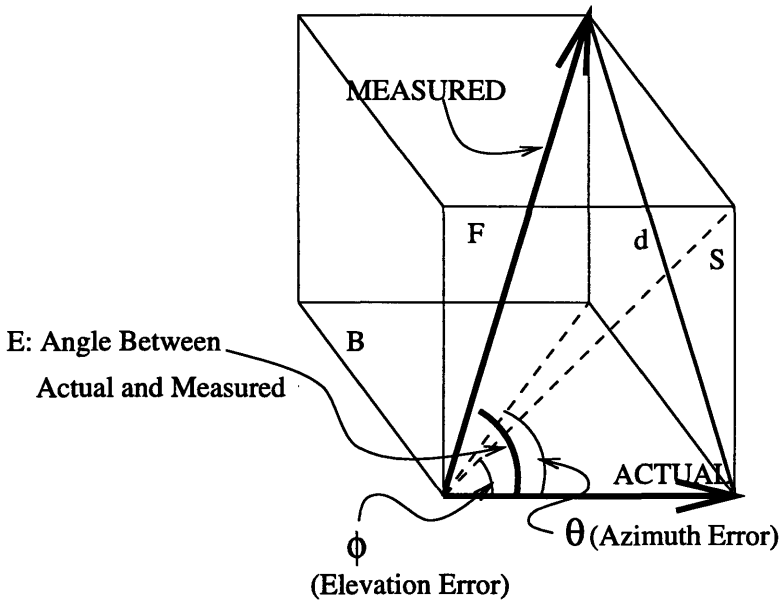

Figure 5-11: Geometric Approach to Find Antenna Pointing Error

The two known values are ϕ, the elevation error, and θ, the azimuth error. The

quantity we are computing for is E, the pointing error, which is the angle between the actual and measured vectors. Examining the bottom-face, labled as B, we assumed the diagonal distance is one and obtained $\cos\theta$ and $\sin\theta$ for the sides adjacent to opposite of the angle θ. This is shown in Figure 5-12.

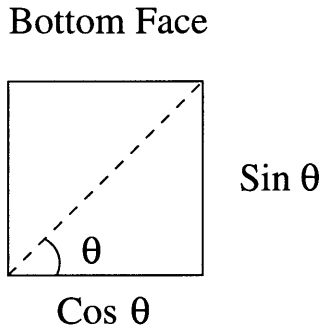

Figure 5-12: Bottom-Face

Next, we proceeded to exam the front-face labled as F. From the previous computation we already know that the side adjacent to ϕ is $\cos\theta$, hence the the side opposite of the angel ϕ, is $\tan\phi\cos\theta$. This is shown is Figure 5-13. Finally, we used the side-face, labled S, whose diagonal is d; it forms a right triangle with the MEASURED and ACTUAL pointing vectors, as shown in Figure 5-14.

First we recognized that,

$$\begin{aligned} d &= \sqrt{\sin^2\theta + \tan^2\phi\cos^2\theta} \\ d &= \cos\theta\sqrt{\tan^2\theta + \tan^2\phi} \end{aligned} \tag{5.1}$$

Once, d is known, then the angle E, pointing error, can be computed using trigonometric identities. We know that,

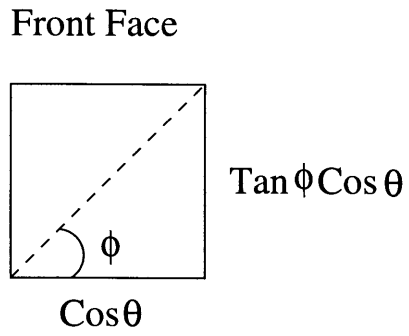

Figure 5-13: Front-Face

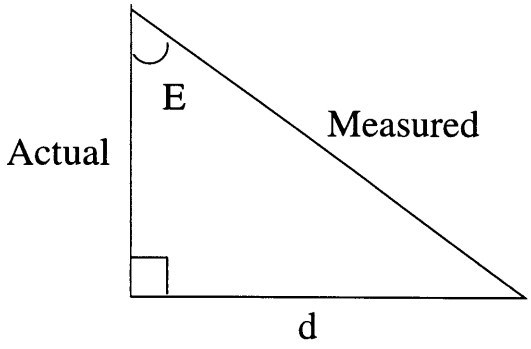

Figure 5-14: Side-Face

$$\begin{aligned} \tan E &= \frac{d}{\cos\theta} \\ \tan E &= \sqrt{\tan^2\theta + \tan^2\phi} \\ E &= \tan^{-1}(\sqrt{\tan^2\theta + \tan^2\phi}) \end{aligned} \tag{5.2}$$

We now have an expression for the pointing error, E, in terms of the elevation error, ϕ, and the azimuth error, θ. We can replace the values ϕ and θ with the computed values of E, and reduce the dimension of the input data set by one. Results from this modification are shown in Table 5-3.

Table 5-3
Results from Testing with Pointing Errors

Stats	Training set	Test set
Accuracy	92.3%	65.7%
MSE	0.106	0.587
Type I Error	3.5%	26.9%
Type II Error	4.2%	7.4%

Unfortunately, these results weren't as good as we expected them to be. Compare them to the results of the original input data set, which is shown in Table 5-1, we concluded that replacing the elevation and azimuth errors with pointing error did not significantly improve the fault detection accuracy. Then, we added the previously computed wind variances information into the data set, and tested the network's performance, its results are shown in Table 5-4.

Compare these results to those shown in Table 5-2, it is clear that the neural network performed worse without having the elevation and azimuth errors in its input data set. From these tests we learned that the elevation and azimuth errors are important inputs that must be kept in the input data set, and having the pointing error information alone is not good enough.

Table 5-4 Results from Testing with Pointing Errors and Wind Variances		
Stats	Training set	Test set
Accuracy	94.1%	79.4%
MSE	0.083	0.381
Type I Error	2.6%	17.5%
Type II Error	3.2%	3.1%

5.3.2 Optimizing Network Architecture

Another method to improve a neural network's performance is to optimize its architecture. It is very important to have a neural network with the appropriate complexity for the problem at hand. Lack of complexity in the network, could cause its inability to model the system. On the other hand, excess complexity could result in an overfitting situation, cause the network to lose its ability to generalize.

One way to vary the complexity of the neural network is to change the number of layers used and the number of neurons in the hidden layer. The number of neurons in the input and output layers are fixed by the problem.

There are theoretical formulas that can be used to calculate the number of neurons needed in order to achieve a desired error tolerance. However these are often not very practical equations, which need messy computations to solve. A more practical method is to determine the optimal network architecture by trial and error.

A series of training experiments were conducted. The only variable in the experiments was the number of hiden layer neurons used. The test was started with the least complex network possible. It was a network consisting of only the input and output layers. We did not expect this simple network to perform well since it probably did not possess the necessary complexity to model the downlink system. Then, an increasing number of hidden layer neurons were added to the neural network. Table 5-5

Table 5-5

Results from Varying the Network Architecture

Neural Networks Architectures		*Training Set*				*Test Set*			
hid. lay.	hid nodes	% corr.	MSE	Type I Error	Type II Error	% corr.	MSE	Type I Error	Type II Error
0	0	86.5	0.207	2.1%	11.4%	78.4	0.293	18.9%	2.7%
1	4	95.6	0.079	1.5%	2.9%	85.7	0.268	10.1%	4.2%
1	6	96.3	0.065	1.8%	1.9%	72.8	0.486	23.8%	3.4%
2	3/3	92.2	0.116	3.5%	4.3%	78.1	0.384	19.4%	2.5%

summarizes the results from these experiments.

From these experiments we can see that there is a clear correlation between the number hidden layer neurons and the detection accuracy. The graph shown in Figure 5-15 sketches out this relationship.

From the graph we can see that when using four hidden layer neurons the best detection accuracy on the test data set was attained. At the same time, an overfitting problem was discovered. Increasing the number of hidden layer neurons from four to six reduced the network's detection accuracy on the test data set, while increased its detection accuracy on the training set. This occurrence is a typical overfitting problem. The added complexity resulted in overfitting of network weights during training coupled with subsequent poor generalization performance on previously unseen input vectors [5]. There is a slight improvement in the network's performance when the same number (i.e., six) hidden neurons were separated into two hidden layers instead of just using one.

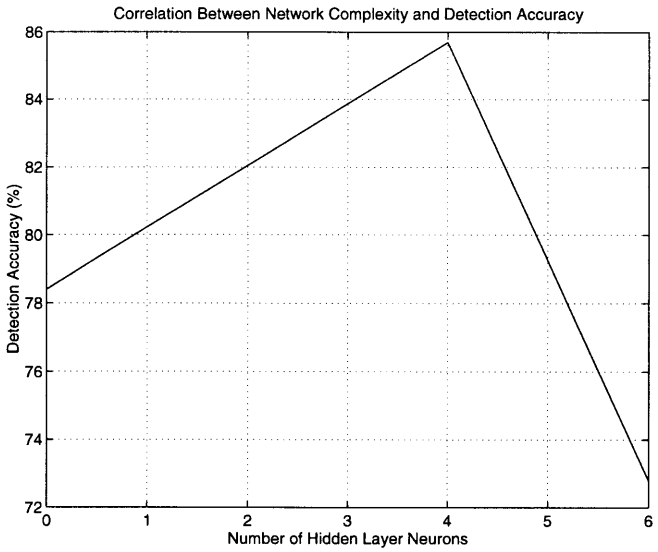

Figure 5-15: Correlation Between Network Complexity and Detection Accuracy

Weight Decaying

One way to optimize network architecture and obtain good generalization ability is to have the network remove non-useful connections during training. A weight that does not change much during the training process will have insignificant effects on the outputs of the neural network. Thus this kind of weights is unimportant and hence should be removed from the network. Such weight removing procedure will help to improve the network's generalization ability [5].

A simple way to remove the unimportant weights during training is called Weight Decaying. The idea is very simple. During the weight training process, weights are adjusted according to the gradient decent method. If we slowly decay each weight in the network, the unimportant weights will eventually decay to close to zero, while the important weights will remain prominent, because they are being constantly reinforced after each training iteration. A small change to the weight-update equations can incorporate weight decaying into the training process [5].

Specifically,

$$w_{ij}^{new} = (1 - \epsilon) w_{ij}^{old} \tag{5.3}$$

where, ϵ, is a very small number that is around 0.001.

After this change was made in the the weight-update process, we tested its effectiveness. An irrelevant input vector was added to the data set, and weights associated with this input vector were decayed close to zeros. Next, we re-trained the network using the added weight decaying feature. Recall from the previous tests, using six hidden layer neurons resulted in an overfitting of the training set, and consequently bad generalization on unseen data in the test set. The same training procedure was repeated, and slightly better results were obtained as shown in Table 5-6.

Table 5-6
Results from Testing with Weight Decaying

Stats	Training set	Test set
Accuracy	94.3%	80%
MSE	0.093	0.318
Type I Error	1.7%	16.5%
Type II Error	4.0%	3.5%

Noticeable improvements were made by adding the weight decaying feature into the original training process. However, there are other more sophisticated methods available for finding efficient architectures. One such method is known as construction algorithms. In weight decaying one starts with a network that is too large for the problem at hand, and gradually prunning away the unimportant weights. A network construction algorithm proceeds in the opposite way, in that it initially starts with a small network, and slowly grow one of the appropriate size [5]. Due to time constraints this path was not explored. We recommend this for future work.

Chapter 6

Conclusions

This thesis described the design, implementation, and use of a multi-layer neural network on the downlink system of the Deep Space Network (DSN). Experiments were conducted to demonstrate the network's ability to perform controlled classical pattern recognition tasks, and to detect anomalous operations in NASA's Deep Space Network's downlink system. Certain issues, such as the optimal network architecture, and the topology of the input space were also studied. This chapter describes the status of the neural network based Downlink Analyzer and concludes with suggestions for future work.

6.1 Status of the Downlink Analyzer

The neural network based Downlink Analyzer is not yet capable of performing real time detection of anomalous operations in the DSN. In other words, the training phases must be separated from the performance phase. Nonetheless, this study shows that a neural network based model is capable of recognizing complex patterns associated with the downlink's system health states when the appropriate architecture is chosen and when a sufficient amount of knowledge about the problem has been built into the the network via the input data file.

The important results of this study are summarized below along with improvements that could possibly lead to higher detection accuracy.

- A generalized DSN model was created. This model helped to gain a more in-depth understanding of the downlink system, and thus enabled some of the knowledge gained to be incorporated into the neural network.

- One way to build knowledge about the problem into the network is through the input space. For example, the variances of Wind Speed and Wind Direction were added into the input data file, and this improved the fault detection accuracy significantly. It is also possible that bringing temporal information into the network could also help improve detection accuracy, since this neural network is incapable of deducing any time dependent information on its own. One possibility is to window the signals to include time lag information. For example, if a high wind speed were the cause of a detected fault, it would take time for this to affect the system. However, this cause and effect situation could be detected if the two events happened to occur in the same time window.

- The number of inputs in the input data file was reduced by combining the elevation and azimuth error into a single pointing error. This seemed reasonable because the the antenna gain is computed only using the pointing error. However, we learned that the elevation and azimuth errors are important feature vectors that should be left in the input data file. Unlike the error signals, the elevation and azimuth themselves are constantly changing in order to track the spacecraft; thus it would be more reasonable to remove elevation and azimuth, rather than the errors from the input data.

- There are many issues associated with finding the optimal network architecture for the given problem, such as how many hidden layers there should be, how they should be organized, and how many neurons are needed for a given problem. A series of experiments were conducted in order to determine the optimal architecture for our particular problem, and found that using a single hidden layer consisting of four neurons worked the best. In the process, an overfitting problem was also discovered when more than four hidden neurons were used. A simple weight decaying method was implemented to improve the

overfitting problem.

6.2 Future Work

The issues studied in this thesis are just a few of the many surrounding the implementation of neural network models. Other related issues that we have not had time to explore are for include whether changing the activation function in the network affects its performance, and if so what sort of activation function should be used. In addition, different weight updating schemes could be explored, such as synchronous vs asynchronous, and deterministic vs stochastic.

We have compiled a list of future work stemmed from this thesis which could be extremely useful to the successful implementation of an automated Downlink Analyzer.

- Build the time dependency information into the network so that it will not have the problem of conflicting data, and will possibly be able to perform fault prediction.

- Complete a model of the DSN's downlink system that is accurate enough to create simulated faults. In other words, we can use this model to generate data that have simulated system faults, and then use the neural network to see if it can detect these faults. This model would be even more useful for fault diagnosis. A neural network can be used to determine which input, or combination of inputs, caused the detected fault. Thus we can alter an input to simulate a particular type of system fault and check to see if the neural network is able to correctly determine which input was the cause.

- Experiment with cost functions different than the one used in this thesis, namely the mean-squared-error (MSE) criterion, which is driven to a local minimum over all the collection training vectors, and is best suited to neural networks that are designed to approximate continuous functions. In the case of "good" or "bad" pattern classification, the target vector components are binary variables

with values of either "0" or "1"; recall that the neuron with the largest output is rounded to be 1, and is interpreted as designating the class determined by the neural network. The MSE cost function is not optimal for this situation because the functions are intrinsically discontinuous, and because the MSE does not always decrease monotonically with the improved classification performance. Other cost functions, such as the classification-figure-of-merit (CFM), is driven by differential learning, seeking only to ensure correct classification rather than trying to accomplish the more difficult task of

modeling a posteriori distribution functions the way MSE-based probabilistic learning does [14].

- Explore the possibilities of a neural network that is able to learn in real time, instead of requiring a training phase that is separated from the performance phase.

These additional studies will help to develop the current Downlink Analyzer into an extremely useful product for monitoring the health state of the DSN's downlink system in the most efficient manner. More importantly, with slight modifications, this neural network based fault detection and/or diagnosis model can be successfully applied to any generic system that possesses similar overall characteristics as the downlink system being modeled in this thesis, provided that we have a good understanding of the problem to be modeled, and we are able to incorporate this knowledge into the neural network through the appropriate input space topology and the optimal network architecture.

Appendix A

MATLAB Source Code

In this appendix, the MATLAB source codes of the pertinent functions are included.

A.1 Functions Responsible for Training the Neural Network

The following four functions are responsible for training the the network using the back-propagation algorithm.

A.1.1 Control and Train

```
% this function performs backpropagation on 1,2 or 3 layer networks
% it allows you to specify the number of neurons in each layer
% it has the MOMENTUM method
% as well as adaptive learning rate
% and normalized gradient vectors
% and batch mode iteration

% this is specifically designed to analyze DLA data.
% where classification is either "good" or "bad"
```

10

```
function [fW1, fW2, fW3, msserror, nt] = adp_train2()
```

```
fname = input('name of the DLA training file:  ');
tname = input('name of the corresponding target file:  ');
num_layer = input('number of layers used in the network:  ');
num_neurons = input('number of neurons in each layer [s1, s2, s3]:  ');
tol = input('error tolerance (rel.  % change):  ');
tol_frq = input('number of consecutive times the error has to stay below the tol level:  ');
method = input('which adaptive Learning Rate method to use (1 to 4):  ');
eg = input('error goal:  ');                                                    20

% INPUTS:
% num_layer: is either 1, 2 or 3
% num_neurons: is a vector that contains the number of neurons per layer
%              not including the inputs.
% tol: error tolerance, which is set as the relative change
%      in the mean-sum-sqr-error from the previous training set
% tol_frq: number of times it stayed at the tol level consecutively
% eg: error goal
% method: which adp lr method to use                                            30

% OUTPUTS:
% fW1, fW2, fW3: the finalized weight matrices for each layer
%                in the network, after trainning has completed
% msserror: is a vector that keeps track the mean sum of squared errors
%           during the course of the trainning process


%%%%%%%%%%%%%%%%
% load training %                                                               40
%    data       %
%%%%%%%%%%%%%%%%

eval(['load ' fname ' -ascii']);
eval(['load ' tname ' -ascii']);

eval(['input_set = ' fname ';']);
eval(['target = ' tname ';']);
```

```
[m, n] = size(input_set)
[mt, nt] = size(target)

%%%%%%%%%%%%%%%%
% normalize      %
% inputs -1 to 1%
%%%%%%%%%%%%%%%%

%normalize DATA -1 to 1
%solving simultaneous eqns
% a*(max) + b = 1
% a*(min) + b = -1
for i = 1:n
   sig = input_set(:,i);
   std_max = max(sig);
   std_min = min(sig);
   a = 2/(std_max - std_min);
   b = 1 - a*std_max;

   input_set(:,i) = input_set(:,i)*a + b;
end


%%%%%%%%%%%%%
% initialize %
% parameters %
%%%%%%%%%%%%%

% init the mean-sum-squared-error to zero
msse = 0;

% init matrix for each layer
% each wt matrix is i by j
% where i is the number of inputs to the layer
% and   j is the number of outputs from the layer
```

```
%initialize wt change matrices to zero
%^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                                                    90
for i = 1:num_layer
  if (i == 1)
   [init_wt1] = init_wt_matrix(n, num_neurons(i))% an extra row of wts
                                    % is added which are wts associated
                                    % with added bias (+1)
   wt1g_chg = zeros(size(init_wt1));
  elseif(i == 2)
    [init_wt2] = init_wt_matrix(num_neurons(i-1), num_neurons(i))
    wt2g_chg = zeros(size(init_wt2));
  else                                                              100
     [init_wt3] = init_wt_matrix(num_neurons(i-1), num_neurons(i));
     wt3g_chg = zeros(size(init_wt3));
  end
end


%%%%%%%%%%%
% User     %
% interface%
%%%%%%%%%%%                                                        110
% create an interface quit button to stop training
global DONE;
DONE = 0; %value of DONE when button is not pressed

figure(1)
quitbutton = uicontrol('String', 'press here to stop', 'Position', [20 20 200 20], 'Style', 'Pus

%%%%%%%%%%%
% TRAINING %                                                       120
```

```
%%%%%%%%%%%

% first we train the entire training set once
% calculate the mean-sum-square-error
% and check if it meets the tol test

% init parameters k and eta, alpha...etc.
% used for adaptive learning rates adjustments

k = 0;                                                                    130
eta = 0.01;
alpha = 0.5;
max_lr = 5;
min_lr = 0.01;
e = 0.001;

%%%%%%%%%%%%%%%
% Train first  %
% with init.   %
% parameters   %                                                          140
%%%%%%%%%%%%%%%

for i = 1:m    % train each input vector in the set
   input_vector = input_set(i, :);
   target_vector = target(i, :);
   [Wt1g, Wt2g, Wt3g, sse] = backprop2(num_layer, input_vector, init_wt1, init_wt2, init_wt3, target_vector, n);
   wt1g_chg = wt1g_chg + Wt1g;
   wt2g_chg = wt2g_chg + Wt2g;
   wt3g_chg = wt3g_chg + Wt3g;
   msse = msse + sse;   %adding up the sse from each input vector          150
end

%%%%%%%%%%%%%%%%%%%%%%%
% Batch mode training %
% normalize the grad. %
% vectors wt_chg      %
```

```
%%%%%%%%%%%%%%%%%%%%%

length = sqrt(sumsqr(wt1g_chg)+sumsqr(wt2g_chg)+sumsqr(wt3g_chg));
if(length ~= 0)                                                    160
  Wt1d = eta * (wt1g_chg/length);
  Wt2d = eta * (wt2g_chg/length);
  Wt3d = eta * (wt3g_chg/length);
end

%%%%%%%%%%%%%%%
% Update weight%
% Parameters   %
%%%%%%%%%%%%%%%
Wt1 = init_wt1 + Wt1d;                                             170
Wt1 = (1 - e) * Wt1;
Wt2 = init_wt2 + Wt2d;
Wt2 = (1 - e) * Wt2;
Wt3 = init_wt3 + Wt3d;
Wt3 = (1 -e) * Wt3;

prev_msse = msse/m;  %divide by the number of input vectors to find the mean sserror
fprintf('BPTRAIN: #%d iteration, MSSE = %e GRAD_MAG = %e\n', 1, prev_msse, length);
lr(1) = eta;
error(1) = prev_msse;                                              180
grad_len(1) = length;

%%%%%%%%%%%%%%%%%%%
% Train the network%
% till the tol is  %
% met, or the user %
% push quit button %
%%%%%%%%%%%%%%%%%%%
rel_diff = tol + 1;
num_train = 2;                                                     190
count = 0;
while ( ((rel_diff > tol) | (count < tol_frq)) & (DONE == 0))
```

```
% initialize parameters to 0
msse = 0;
wt1g_chg = zeros(size(wt1g_chg));
wt2g_chg = zeros(size(wt2g_chg));
wt3g_chg = zeros(size(wt3g_chg));

if (rel_diff < tol)  %keeps track the # of times
   count = count + 1; % that it has met the tolerance                    200
else
   count = 0;
end

%to check if Learning rate need to be adjusted

if (k >= 3) % after at 3 consecutive good steps
   eta = Adp_lr(method, eta, chg_in_error, min_lr, max_lr);
end
lr(num_train) = eta;                                                     210

%%%%%%%%%%%%%%%%%%%%
% backprop. on each %
%  input pattern    %
% the training set  %
%%%%%%%%%%%%%%%%%%%%

   for i = 1:m    % train each input vector in the set
      input_vector = input_set(i, :);
      target_vector = target(i, :);                                      220
      [Wt1g, Wt2g, Wt3g, sse] = backprop2(num_layer, input_vector, Wt1, Wt2, Wt3, target_vector, n);
      wt1g_chg = wt1g_chg + Wt1g;
      wt2g_chg = wt2g_chg + Wt2g;
      wt3g_chg = wt3g_chg + Wt3g;
      msse = msse + sse;
   end
```

```
%%%%%%%%%%%%%%%%%%%%%
% Batch mode training %                                          230
% normalize the grad. %
% vectors wt_chg      %
%%%%%%%%%%%%%%%%%%%%%

length = sqrt(sumsqr(wt1g_chg)+sumsqr(wt2g_chg)+sumsqr(wt3g_chg));
if(length ~= 0)
  Wt1d = eta * (wt1g_chg/length) + alpha * Wt1d;
  Wt2d = eta * (wt2g_chg/length) + alpha * Wt2d;
  Wt3d = eta * (wt3g_chg/length) + alpha * Wt3d;
end                                                              240
grad_len(num_train) = length;

%%%%%%%%%%%%%%%
% Update weight%
% Parameters   %
%%%%%%%%%%%%%%%
Wt1 = Wt1 + Wt1d;
Wt1 = (1 - e) * Wt1;
Wt2 = Wt2 + Wt2d;
Wt2 = (1 - e) * Wt2;                                             250
Wt3 = Wt3 + Wt3d;
Wt3 = (1 - e) * Wt3;

%%%%%%%%%%%%%%%%
% if cost funct.%
% increased     %
%%%%%%%%%%%%%%%%
    msse = msse/m;
    chg_in_error = msse - prev_msse;
    if (chg_in_error > 0) % cost function increased, so eta should be decreased   260
      k = 0;
      alpha = 0;  % set momentum coeff to 0

      % undo the weight changes
```

```
      if (num_train == 2)
        Wt1 = init_wt1;
        Wt2 = init_wt2;
        Wt3 = init_wt3;
      else
        Wt1 = good_wt1;                                          270
        Wt2 = good_wt2;
        wt3 = good_wt3;
      end


    eta = Adp_lr(method, eta, chg_in_error, min_lr, max_lr);

%%%%%%%%%%%%%%%%
% if cost funct.%
% decreased     %                                              280
%%%%%%%%%%%%%%%%
    else
      k = k + 1; % keep track of how consistently the cost function is decreasing
      alpha = 0.9;

      % only keep those good wts and errors
      good_wt1 = Wt1 - Wt1d;
      good_wt2 = Wt2 - Wt2d;
      good_wt3 = Wt3 - Wt3d;
      error(num_train) = msse;                                   290

%%%%%%%%%%%%%%%%
% cal. & graph %
% the cost func%
% and grad len %
%%%%%%%%%%%%%%%%
      figure(1);
      errplot(error, eg);
      drawnow;
                                                                 300
```

```
        figure(2)
        gradplot(grad_len, 0);

%%%%%%%%%%%%%%%
% cal. & graph %
% learning rt. %
%%%%%%%%%%%%%%%
      % figure(3)
       % lrplot(lr, max_lr, num_train);
       %drawnow;                                                          310
       num_train = num_train + 1;
       rel_diff = (abs((chg_in_error)/prev_msse)) * 100;
       prev_msse = msse;
       fprintf('BPTRAIN: #%d iteration, MSSE = %e, GRAD_MAG = %e PERCENT_CHG = %f\n', num_trai

     end
end


%%%%%%%%%%%%%%%                                                          320
% Storing the  %
%   Results    %
%%%%%%%%%%%%%%%

fW1 = Wt1;
fW2 = Wt2;
fW3 = Wt3;
msserror = error;
nt = num_train;
                                                                          330
% store the resulting weight matrices in separate files
% so can be used later to test the network

[r1, c1] = size(fW1);
[r2, c2] = size(fW2);
[r3, c3] = size(fW3);
```

```
% also store the weights in a file called wts.dat

fid1 = fopen('wt1.dat', 'w');                                          340
fid2 = fopen('wt2.dat', 'w');
fid3 = fopen('wt3.dat', 'w');

for i = 1:r1
   fprintf(fid1, '\n');
   for j = 1:c1
      fprintf(fid1, '%f\t', fW1(i,j));
   end
end
for i = 1:r2                                                           350
   fprintf(fid2, '\n');
   for j = 1:c2
      fprintf(fid2, '%f\t',fW2(i,j));
   end
end
for i = 1:r3
   fprintf(fid3, '\n');
   for j = 1:c3
      fprintf(fid3, '%f\t', fW3(i,j));
   end                                                                 360
end

fclose(fid1);
fclose(fid2);
fclose(fid3);




                                                                       370
```

A.1.2 Perform Back-propagation Algorithm

```
% this function performs backpropagation method on a one, two,
% or maximum three-layer Neural Network. It also uses MOMENTUM
% the gradient vectors are normalized

function [Wt1g, Wt2g, Wt3g, SSE] = backprop(num_layer, input_vector, w1, w2, w3, target, n)
% INPUTS:
% num_layer: is either 1, 2 or 3
% input_vector: is a row vector from the training set
% target: the target output
% w1,w2,w3: wt matrices for each layer from the previous training                    10
% n: number of components in each input vector that is
%    inputs to the first layer

% OUTPUTS:
% Wt1g, Wt2g, Wt3g: gradient vectors after current training
% SSE: sum squared error (i.e. (T-A)^2)

%%%%%%%%%%%%%%%%%%%%%%
% find the ouputs for %
% for each layer      %                                                            20
%%%%%%%%%%%%%%%%%%%%%%
nl = num_layer;
for i = 1:num_layer
   if (i == 1)
      input_vector(n+1) = 1;
      [out] = NNout2(input_vector, w1);
      NN_out = out;
      s = size(out,2);
      out(s+1) = 1; % adding the bias (+1) to the next layer
   elseif (i == 2)                                                                  30
      hidden1 = NN_out';
```

```
    [out] = NNout2(out, w2); % output of the first layer is now the input
    NN_out = out;
    s = size(out,2);
    out(s+1) = 1;
  elseif(i == 3)
    hidden2 = NN_out';
    [out] = NNout2(out, w3);
    NN_out = out;
  end                                                              40
end
%%%%%%%%%%%%%%%%%%%
%  SINGLE LAYER    %
%  NETWORK         %
%%%%%%%%%%%%%%%%%%%

if (num_layer == 1)
   [delta, wt1g, sse] = Delta_Rule(target, NN_out, input_vector);

%%%%%%%%%%%%%%%%%%%                                              50
%   MULTI_LAYER   %
%   NETWORK       %
%%%%%%%%%%%%%%%%%%%
else  % multi_layer network training

   % calculate the error between the network output and the desired output
   % adjust the weights of the NN in a way that minimizes the error
   % delta is the adjustment for each neuron in the output layer;

%%%%%%%%%%%%%%%%%%%                                              60
% Adjust weights %
% of output layer%
%%%%%%%%%%%%%%%%%%%

   % first calculate the output layer weight adjustments
   % using the output from the previous hidden layer
```

```
hidden_layer = num_layer - 1;
if (hidden_layer == 1)
   prev_out = hidden1;                                                    70
   [delta, wt_grad, sse] = WtAdj2(target, NN_out, prev_out);
   wt2g = wt_grad;
else
   prev_out = hidden2;
   [delta, wt_grad, sse] = WtAdj2(target, NN_out, prev_out);
   wt3g = wt_grad;
end

%%%%%%%%%%%%%%%%%%%%
% Adjust Weights of %                                                    80
% the hidden layer  %
%%%%%%%%%%%%%%%%%%%%

 % calls on HwtAdj function to calculate the deltas
 % for the hidden layer
 % find the weights adjustments from previous layer
 while(num_layer > 1)

    % adjusting the hidden layer next to the inputs
    if (hidden_layer == 1)                                                90
      current_out = hidden1;
      prev_out = input_vector(1:n);
      [delta, hwt_grad] = HwtAdj2(delta, w2, current_out, prev_out');
      wt1g = hwt_grad;

    elseif(hidden_layer == 2) % adjusting the hidden layer next to the output layer
      current_out = hidden2;
      prev_out = hidden1;
      [delta, hwt_grad] = HwtAdj2(delta, w3, current_out, prev_out);
      wt2g = hwt_grad;                                                    100
    end

    num_layer = num_layer - 1;
```

```
        hidden_layer = hidden_layer - 1;
    end
end

Wt1g = wt1g;
Wt2g = wt2g;
Wt3g = wt3g;
SSE = sse;
```

A.1.3 Adjusting Weights for a Single Layer Network

```
% this function takes in the network outputs and the target output
% and returns the correct weight adjustments for the output layer only
% using the momentum method which involves adding a to the weight adjustment
% that is proportional to the amount of the previous weight adjustment

%%%%%%%%%%%%%%%%%%
% this is for a  %
% single layer   %
% perceptron     %
%%%%%%%%%%%%%%%%%%

function [delta, wt_grad, sserror] = Delta_Rule(target, NN_out, prev_out)
%INPUTS:
% target: is the target output
% NN_out: ouput of the current layer k
% prev_out: outputs of previous layer j
```

```
%OUTPUTS:
% delta = TARGET - OUT
% wt_grad:  WT_GRADpq,k = DELTAq,k * OUTp,j
%         where: p is the neuron from previous layer j
%                q is the neuron of the current layer k
%                OUTp,j is the ouput of neuron p from layer j
%         wt_grad will be normalized to unit vectors
% sserror: is the sum squared error

n = size(NN_out, 2);
pn = size(prev_out,2);

% there is one delta for each neuron in a layer
% n is the number of neurons in each layer k
delta = target - NN_out;
sserror = sumsqr(delta);

% once we have obtained delta for each neuron
% we have to now calculate the weight change

for q = 1:n
   wg(:,q) = delta(q) * prev_out';
end

% this is the now the correct weight change for this training
wt_grad = wg;
```

A.1.4 Adjusting Weights for a Multi-layer Network

For a multi-layer network, separate functions are needed for adjusting the weight parameters in the hidden layers and the output layer.

Adjusting Weights in the Hidden Layers

```
% this function calculates the delta values for the hidden layers in the NN
% using the momentum method

function [h_del, wt_grad] = HwtAdj2(delta, back_wts, outputs, prev_out)
%INPUTS:
% delta: delta from the layer ahead (layer k) (DELq,k)
% back_wts: weight matrix connected to the layer ahead (Wpq,k)
% outputs: outputs of the current hidden layer (OUTp,j)
% prev_out: outputs from the previous layer

%OUTPUTS:
% h_del: delta for this hidden layer j (DELp,j = OUTp,j(1-OUTp,j)SUMq(DELq,k * Wpq,k))
% wt_grad: wt gradients which are later normalized to unit vectors

n = size(outputs, 1); %n is the number of outputs for the hidden layer
deriv = actDeriv(outputs); %derivative of the activation function OUTp,j(1-OUTp,j)


% info on previous outputs
% a bias of +1 attached
pn = size(prev_out,1);
prev_out(pn+1) = 1;

% first calculate delta for the hidden layer
% from the ouput layer's delta or the previous hidden layer's delta
% equivalent as finding the sum: SUMq(DELq,k * Wpq,k)

for p = 1:n
   f = back_wts(p, :) * delta'; %multiply all the wts connected from
```

```
                    %NEURONp,j to NEURONq,k by DELq,k                    30
                    %and sum them (i.e. W11 * DEL1 + W12 * DEl1..)
  hd(p) = deriv(p) * f;  %multiply the derivative of the activation function
end                 %evaluated at OUTp,j

% the rest is the same as in the output layer
for j = 1:n
      wg(:,j) = hd(j) * prev_out;
end

h_del = hd;                                                              40
wt_grad = wg;
```

50

Adjusting Weights in the Output Layer

```
% this function takes in the network outputs and the target output
% and returns the correct weight adjustments for the output layer only
% using the momentum method which involves adding a to the weight adjustment
% that is proportional to the amount of the previous weight adjustment

function [delta, wt_grad, sserror] = WtAdj2(target, NN_out, prev_out)
%INPUTS:
% target: is the target output
% NN_out: ouput of the current layer k
% prev_out: outputs of previous layer j                                   10

%OUTPUTS:
```

```
% delta = OUT(1-OUT)(TARGET - OUT)
%       where OUT(1-OUT) is the derivative of the activation function
%       evaluated at OUTPUT of the Nnet
% wt_grad:  WT_GRADpq,k = DELTAq,k * OUTp,j
%         where: p is the neuron from previous layer j
%                q is the neuron of the current layer k
%                OUTp,j is the ouput of neuron p from layer j
%         it will be normalized later to unit vectors                    20
% sserror: is the sum squared error

n = size(NN_out, 2);
deriv = actDeriv(NN_out);

% for training the bias weight
% OUTp,j is now +1, +1 has added to bias the neuron
pn = size(prev_out,1);
prev_out(pn+1) = 1;
                                                                         30
% there is one delta for each neuron in a layer
% n is the number of neurons in each layer k

e = target - NN_out;
delta = deriv .* e;
sserror = sumsqr(e);

% once we have obtained delta for each neuron
% we have to now calculate the weight change
                                                                         40
for q = 1:n
      wg(:,q) = delta(q) * prev_out;
end

% this is the now the correct weight change for this training
wt_grad = wg;
```

A.2 Testing the Network

Once the network is fully trained, the following program is used to perform fault detection on the test data set.

```
% After the neural network has been properly trained.
% it will be tested using real Downlink System NORM_DATA

function test
global DATA;

input_set = DATA(:, 2:12); %taking 11 feature vectors

%%%%%%%%%%%%%%%%
% Converting Az.%
% & Elv. Err to %
% Pointing Err. %
%%%%%%%%%%%%%%%%

AzErr = input_set(:,3);
ElvErr = input_set(:,5);

% convert them from degrees to radians

AzErr = AzErr/180*pi;
ElvErr = ElvErr/180*pi;

PtErr = atan(sqrt( (tan(AzErr)).^2 + (tan(ElvErr)).^2 ) );
PtErr = PtErr/pi*180;

tdata(:,1:2) = input_set(:,1:2);
tdata(:,3) = input_set(:,4);
```

```
tdata(:,4) = PtErr;
tdata(:,5:8) = input_set(:,6:9);

[m,n] = size(tdata);

%normalize DATA -1 to 1
%solving simultaneous eqns
% a*(max) + b = 1
% a*(min) + b = -1
for i = 1:n
   std_sig = tdata(:,i);
   std_max = max(std_sig);
   std_min = min(std_sig);
   a = 2/(std_max - std_min);
   b = 1 - a*std_max;

   norm_data(:,i) = tdata(:,i)*a + b;
end

tname = input('input name of the target file:  ');
num_layer = input('how many layers to use:  ');

% first we train the network with a randomly generated trainning norm_data set
% and find out the final weights

for i = 1:num_layer
   filename = sprintf('wt%d.dat', i);
   load(filename);
end

eval(['load ' tname ' -ascii']);
eval(['target = ' tname ';']);

[m,n] = size(norm_data)
[mt, nt] = size(target)
nb = n+1;
```

```
class = target(:,1);
plot(class, 'go');
ylabel('BAD(0)                                                    GOOD(1)');
title('Neural Network Simulation Results');
grid;
hold on;                                                                    70

fprintf('\n\n%8s %8s %8s %15s\n', 'Number', 'Health', 'Model', 'Differential');
g = 0;
b = 0;
t1 = 0;
t2 = 0;

% classify each component in the test norm_data set
for i = 1:m
   test_input = norm_data(i,:);                                             80
   real = target(i,:);
   test_input(nb) = 1;
   for j = 1:num_layer
      if (j == 1)
         [out] = NNout2(test_input, wt1);
         NN_out = out;
         s = size(out,2);
         out(s+1) = 1; % adding the .datbias
      elseif (j == 2)
         hidden1 = out';                                                    90
         [out] = NNout2(out, wt2);
         NN_out = out;
         s = size(out,2);
         out(s+1) = 1;
      elseif(j == 3)
         hidden2 = NN_out';
         [out] = NNout2(out, wt3);
         NN_out = out;
      end
```

```
    end                                                                    100
    sse(i) = sumsqr(real - NN_out);

    % pick the maximum of the the output components (i.e. from each
    % neuron of the output layer) and set that to "1", and set the rest
    % to "0".
    model_output = out_conversion(NN_out);
    r(i,:) = model_output;
    differential = abs(NN_out(1) - NN_out(2));
    model_state = state_name(model_output);
    health = state_name(real);                                             110

    if ( (strcmp(health, 'Good')) == 1);
      g = g + 1;
      if ( (strcmp(health, model_state)) == 0)
        t1 = t1 + 1;
      end
    else
      b = b + 1;
      if ( (strcmp(health, model_state)) == 0)
        t2 = t2 + 1;                                                       120
      end
    end
    fprintf('%8d %8s %8s %15f\n', i, health, model_state, differential);
end

c = r(:,1) - 0.1;
plot(c, 'r*');
hold off

w = t1 + t2;                                                               130
acur = 1 - w/m;

t1g = t1/g * 100;
t2b = t2/b * 100;
msse = (sum(sse))/m;
```

```
fprintf('This data set contains %d Good states, and %d Bad states\n', g, b)
fprintf('Classification Accuracy:  %8f\n', acur);
fprintf('Mean Squared Error:  %8f\n', msse);
fprintf('TypeI Errors(false alarms):  %d, which is %f percent of all the Good States\n', t1,
fprintf('TypeII Errors(lack of detection of a fault):  %d which is %f percent of all the Ba
```

A.3 Graphic User Interface

Some graphic user interface features are also included. One such feature is used during the training phase, where it automatically updates the training process, namely it shows graphically how the mean-squared-error is converging.

```
function h2=errplot(e,g,h)
%ERRPLOT Plot network sum-squared error vs epochs.
%
%       ERRPLOT(E,G)
%         E - Row vector of error values.
%         G - Error goal.
%       Returns (optionally) handle to error curve in plot.
%
%       ERRPLOT(E,G,H)
%         H - Handle returned by previous call to PLOTERR.
%       Deletes old error curve H, and plots new one.

if nargin < 1,error('Not enough arguments'), end

iterations = length(e)-1;
t = sprintf('Mean-Sum-Squared Network Error for %g Iterations', iterations);
```

```
% BACKWARD COMPATIBILITY FOR NNT 1.0
% Convert PLOTERR(E,T) -> PLOTERR(E)
nargin2  = nargin;                                                    20
if nargin2 == 2
  if isstr(g)
    t = g;
        nargin2 = 1;
  end
end

if nargin2 < 3
  newplot;
  if nargin2 == 2                                                     30
    plot([0 999999],[g g],'r:',0,g*0.9,'.b')
  end
  xlabel('Iteration')
  ylabel('Mean-Sum-Squared Error')
  title(t)
  set(gca,'box','on')
else
  delete(h);
end
                                                                      40
hold on
e = e + eps;
H = plot(0:iterations,e);
title(t)
hold off

set(gca,'xlim',[0 iterations+eps]);
set(gca,'yscale','log');
drawnow
                                                                      50
if nargout == 1
  h2 = H;
end
```

A.3.1 Anlaysis Tool and Simulation Results

Another feature is used to demonstrate simulation results, and can be used as a simple statistical analysis tool on the input signals.A typical analysis tool looks like the following:

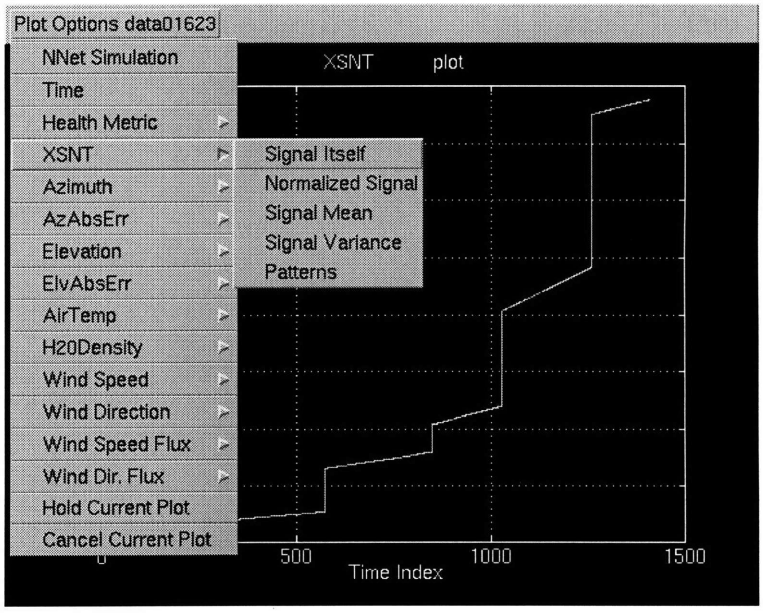

Figure A-1: Menu for the Analysis Tool

function plot_menu(fname)

global DATA;

global SIG_NAMES;

global SIG_NAMES2;

global WIN_SIZE;

```
WIN_SIZE = 10;
eval(['load ' fname ' -ascii']);
eval(['DATA = ' fname ';']);
menu_title = sprintf('Plot Options %s', fname);

SIG_NAMES = str2mat('Time', 'XSNT', 'Azimuth', 'AzAbserr', 'Elevation', 'ElvAbsErr', 'AirTemp'
SIG_NAMES2 = str2mat('Wind Speed Flux', 'Wind Direction Flux', 'Health Metric');

% create menus
plot_opt = uimenu(gcf, 'Label', menu_title);
plot_nn = uimenu(plot_opt, 'Label', 'NNet Simulation', 'Callback', 'test;', 'Separator', 'on');
plot_tm = uimenu(plot_opt, 'Label', 'Time', 'Callback', 'plot_sig(1);', 'Separator', 'on');
plot_hm = uimenu(plot_opt, 'Label', 'Health Metric', 'Separator', 'on');
plot_xsnt = uimenu(plot_opt, 'Label', 'XSNT', 'Separator', 'on');
plot_az = uimenu(plot_opt, 'Label', 'Azimuth', 'Separator', 'on');
plot_azerr = uimenu(plot_opt, 'Label', 'AzAbsErr', 'Separator', 'on');
plot_elv = uimenu(plot_opt, 'Label', 'Elevation', 'Separator', 'on');
plot_elverr = uimenu(plot_opt, 'Label', 'ElvAbsErr', 'Separator', 'on');
plot_at = uimenu(plot_opt, 'Label', 'AirTemp', 'Separator', 'on');
plot_den = uimenu(plot_opt, 'Label', 'H20Density', 'Separator', 'on');
plot_ws = uimenu(plot_opt, 'Label', 'Wind Speed', 'Separator', 'on');
plot_wd = uimenu(plot_opt, 'Label', 'Wind Direction', 'Separator', 'on');
plot_wsf = uimenu(plot_opt, 'Label', 'Wind Speed Flux', 'Separator', 'on');
plot_wdf = uimenu(plot_opt, 'Label', 'Wind Dir.  Flux', 'Separator', 'on');
plot_stay = uimenu(plot_opt, 'Label', 'Hold Current Plot', 'Callback', 'hold on', 'Separator', 'on'
plot_cancel = uimenu(plot_opt, 'Label', 'Cancel Current Plot', 'Callback', 'hold off', 'Separator',

%create submenus
signal = uimenu(plot_hm, 'Label', 'Signal Itself', 'Callback', 'plot_hsig(13);');
norm_sig = uimenu(plot_hm, 'Label', 'Normalized Signal', 'Callback', 'plot_nhsig(13);');
sig_mean = uimenu(plot_hm, 'Label', 'Signal Mean', 'Callback', 'mnvar(13,1);');
sig_var = uimenu(plot_hm, 'Label', 'Signal Variance', 'Callback', 'mnvar(13,0);');
sig_pat = uimenu(plot_hm, 'Label', 'Patterns', 'Callback', 'pattern(13,13);');

signal = uimenu(plot_xsnt, 'Label', 'Signal Itself', 'Callback', 'plot_sig(2);');
```

```
norm_sig = uimenu(plot_xsnt, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(2);');
sig_mean = uimenu(plot_xsnt, 'Label', 'Signal Mean', 'Callback', 'mnvar(2,1);');
sig_var = uimenu(plot_xsnt, 'Label', 'Signal Variance', 'Callback', 'mnvar(2,0);');
sig_pat = uimenu(plot_xsnt, 'Label', 'Patterns', 'Callback', 'pattern(13,2);');

signal = uimenu(plot_az, 'Label', 'Signal Itself', 'Callback', 'plot_sig(3);');
norm_sig = uimenu(plot_az, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(3);');
sig_mean = uimenu(plot_az, 'Label', 'Signal Mean', 'Callback', 'mnvar(3,1);');
sig_var = uimenu(plot_az, 'Label', 'Signal Variance', 'Callback', 'mnvar(3,0);');
sig_pat = uimenu(plot_az, 'Label', 'Patterns', 'Callback', 'pattern(13,3);');

signal = uimenu(plot_azerr, 'Label', 'Signal Itself', 'Callback', 'plot_sig(4);');
norm_sig = uimenu(plot_azerr, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(4);');
sig_mean = uimenu(plot_azerr, 'Label', 'Signal Mean', 'Callback', 'mnvar(4,1);');
sig_var = uimenu(plot_azerr, 'Label', 'Signal Variance', 'Callback', 'mnvar(4,0);');
sig_pat = uimenu(plot_azerr, 'Label', 'Patterns', 'Callback', 'pattern(13,4);');

signal = uimenu(plot_elv, 'Label', 'Signal Itself', 'Callback', 'plot_sig(5);');
norm_sig = uimenu(plot_elv, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(5);');
sig_mean = uimenu(plot_elv, 'Label', 'Signal Mean', 'Callback', 'mnvar(5,1);');
sig_var = uimenu(plot_elv, 'Label', 'Signal Variance', 'Callback', 'mnvar(5,0);');
sig_pat = uimenu(plot_elv, 'Label', 'Patterns', 'Callback', 'pattern(13,5);');

signal = uimenu(plot_elverr, 'Label', 'Signal Itself', 'Callback', 'plot_sig(6);');
norm_sig = uimenu(plot_elverr, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(6);');
sig_mean = uimenu(plot_elverr, 'Label', 'Signal Mean', 'Callback', 'mnvar(6,1);');
sig_var = uimenu(plot_elverr, 'Label', 'Signal Variance', 'Callback', 'mnvar(6,0);');
sig_pat = uimenu(plot_elverr, 'Label', 'Patterns', 'Callback', 'pattern(13,6);');

signal = uimenu(plot_at, 'Label', 'Signal Itself', 'Callback', 'plot_sig(7);');
norm_sig = uimenu(plot_at, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(7);');
sig_mean = uimenu(plot_at, 'Label', 'Signal Mean', 'Callback', 'mnvar(7,1);');
sig_var = uimenu(plot_at, 'Label', 'Signal Variance', 'Callback', 'mnvar(7, 0);');
sig_pat = uimenu(plot_at, 'Label', 'Patterns', 'Callback', 'pattern(13,7);');

signal = uimenu(plot_den, 'Label', 'Signal Itself', 'Callback', 'plot_sig(8);');
```

```
norm_sig = uimenu(plot_den, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(8);');
sig_mean = uimenu(plot_den, 'Label', 'Signal Mean', 'Callback', 'mnvar(8,1);');
sig_var = uimenu(plot_den, 'Label', 'Signal Variance', 'Callback', 'mnvar(8,0);');
sig_pat = uimenu(plot_den, 'Label', 'Patterns', 'Callback', 'pattern(13,8);');

signal = uimenu(plot_ws, 'Label', 'Signal Itself', 'Callback', 'plot_sig(9);');
norm_sig = uimenu(plot_ws, 'Label', 'Normalized Signal', 'Callback', 'plot_sig(9);');
sig_mean = uimenu(plot_ws, 'Label', 'Signal Mean', 'Callback', 'mnvar(9,1);');
sig_var = uimenu(plot_ws, 'Label', 'Signal Variance', 'Callback', 'mnvar(9,0);');
sig_pat = uimenu(plot_ws, 'Label', 'Patterns', 'Callback', 'pattern(13,9);');

signal = uimenu(plot_wd, 'Label', 'Signal Itself', 'Callback', 'plot_sig(10);');
norm_sig = uimenu(plot_wd, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(10)');
sig_mean = uimenu(plot_wd, 'Label', 'Signal Mean', 'Callback', 'mnvar(10,1);');
sig_var = uimenu(plot_wd, 'Label', 'Signal Variance', 'Callback', 'mnvar(10, 0);');
sig_pat = uimenu(plot_wd, 'Label', 'Patterns', 'Callback', 'pattern(13,10);');

signal = uimenu(plot_wsf, 'Label', 'Signal Itself', 'Callback', 'plot_sig(11);');
norm_sig = uimenu(plot_wsf, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(11);');
sig_mean = uimenu(plot_wsf, 'Label', 'Signal Mean', 'Callback', 'mnvar(11,1);');
sig_var = uimenu(plot_wsf, 'Label', 'Signal Variance', 'Callback', 'mnvar(11, 0);');
sig_pat = uimenu(plot_wsf, 'Label', 'Patterns', 'Callback', 'pattern(13,11);');

signal = uimenu(plot_wdf, 'Label', 'Signal Itself', 'Callback', 'plot_sig(12);');
norm_sig = uimenu(plot_wdf, 'Label', 'Normalized Signal', 'Callback', 'plot_nsig(12);');
sig_mean = uimenu(plot_wdf, 'Label', 'Signal Mean', 'Callback', 'mnvar(12,1);');
sig_var = uimenu(plot_wdf, 'Label', 'Signal Variance', 'Callback', 'mnvar(12, 0);');
sig_pat = uimenu(plot_wdf, 'Label', 'Patterns', 'Callback', 'pattern(13,12);');
```

Bibliography

[1] A. Bernieri, M. D'Apuzzo, L. Sansone, and M. Savastano. A neural network approach for identification and fault diagnosis on dynamic systems. *Proceedings of IEEE*, 43(6), December 1994.

[2] J. Y. Fan, M. Nikolaou, and R. E. White. An approach to fault diagnosis of chemical processes via neural networks. *AIChE Journal*, 39(1), January 1993.

[3] W. Gawronski. Wind gust models derived from field data. The Telecommunications and Data Acquisition Progress Report 42-123, Jet Propulsion Laboratory, 1995.

[4] J. B. Hampshire and D. A. Watola. Automated downlink analysis for the deep space network. The Telecommunications and Data Acquisition Progess Report 42-125. To Appear.

[5] J. B. Hampshire and D. A. Watola. Diagnosing and correcting system anomalies with a robust classifier. *IEEE Proceedings of the 1996 International Conference on Acoustics, Speech, and Signal Processing*, May 1996.

[6] J. B. Hampshire, D. A. Watola, and S. A. Townes. *Automated Downlink Analysis.* Jet Propulsion Laboratory, September 1994. Submission for Special Signals and Systems Issue of the IEEE Proceedings.

[7] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction To the Theory of Neural Computation.* Addison-Wesley Publishing Company, Redwood City, CA, 1991.

[8] M. L. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1969.

[9] T. A. Rebold, A. Kwok, and G. E. Wood. The mars observer ka-band link experiment. The Telecommunications and Data Acquisition Progress Report 42-117, Jet Propulsion Laboratory, 1994.

[10] P. Smyth and J. Mellstrom. Initial results on fault diagnosis of dsn antenna control assemblies using pattern recognition techniques. The Telecommunications and Data Acquisition Progress Report 42-101, Jet Propulsion Laboratory, 1990.

[11] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alko. Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59:257–263, 1988.

[12] Philip D. Wasserman. *Neural Computing Theory and Practice.* Van Nostrand Reinhold, New York, NY, 1989.

[13] Kajiro Watanabe, Seiichi Hirota, Liya Hou, and D. M. Himmelblau. Diagnosis of multiple simultaneous fault via hierarchical artificial neural networks. *AIChE Journal*, 40(5), May 1994.

[14] Dave Watola. *Overview of Downlink Analyzer Neural Nework Technology.* JPL IOM 331.1-95-047, November 1995.

[15] Dave Watola. *Overview of the Downlink Analyzer.* JPL IOM 331.1-95-045, November 1995.

[16] Paul J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of IEEE*, 78(10), October 1990.

[17] Bernard Widrow and Michael A. Lehr. 30 years of adaptive neural networks: Perceptron, madaline, and backpropagation. *Proceedings of IEEE*, 78(9), September 1990.

[18] Prof. Victor Zhu. Lecture Handouts for 6.345 Automatic Speech Recognition, M.I.T. Spring, 1995.