MODIFIED BACKWARD ERROR PROPAGATION FOR TACTICAL TARGET RECOGNITION

THESIS

Charles C. Piazza
Captain, USAF

AFIT/GE/ENG/88D-36

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GE/ENG/88D-36

MODIFIED BACKWARD ERROR PROPAGATION FOR TACTICAL TARGET RECOGNITION

THESIS

Charles C. Piazza
Captain, USAF

AFIT/GE/ENG/88D-36

AFIT/GE/ENG/88D-36

MODIFIED BACKWARD ERROR PROPAGATION FOR TACTICAL TARGET RECOGNITION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Charles C. Piazza, B.S.
Captain, USAF

December 1988

Acknowledgments

There were many people behind the scenes of this research effort, who played key roles in the development of my thesis. First and foremost, is my wife [redacted] With her complete understanding and support, and her many words of encouragement, my potential was realized and personal goals achieved. I am truly indebted to Dr. Steven K. Rogers and Dr. Mark E. Oxley for their seemingly unlimited amount of knowledge, advice, and time afforded to me. Special thanks to Dr. Rogers, for instilling within me his great enthusiasm for higher learning.

I would also like to thank Dr. Mathew Kabrisky for his assistance and input, and especially for his wit and wonderful view of the world in which we live. Many thanks to Captain Mike Roggemann for his efforts in providing me with the forward looking infrared imagery feature vectors, and for our many discussions.

Table of Contents

List of Figures

Figure

List of Tables

Table

Abstract

This thesis explores a new approach to the classification of tactical targets using a new biologically-based neural network. The targets of interest were generated from doppler imagery and forward looking infrared imagery, and consisted of tanks, trucks, armored personnel carriers, jeeps and petroleum, oil, and lubricant tankers. Each target was described by feature vectors, such as normalized moment invariants. The features were generated from the imagery using a segmenting process. These feature vectors were used as the input to a neural network classifier for tactical target recognition.

The neural network consisted of a multilayer perceptron architecture, employing a backward error propagation learning algorithm. The minimization technique used was an approximation to Newton's method. This second order algorithm is a generalized version of well known first order techniques, i.e., gradient of steepest descent and momentum methods. Classification using both first and second order techniques was performed, with comparisons drawn.

Modified Backward Error Propagation
for Tactical Target Recognition

1. Introduction

1.1. Historical Background

In recent years there has been an enormous increase in the interest of artificial neural networks (ANNs) in a variety of disciplines. One of the reasons behind this renewed interest, is ANNs may provide a solution to the problem of machine interpretation of image and voice patterns; a solution that has thus far eluded the digital computer. Therefore, it is no great wonder that ANNs have sparked the interest of scientific and engineering groups within the military community. From a military aspect, if machines can be taught to learn and recognize patterns, then it would be possible to realize an autonomous weapons system. A piloted aircraft could deliver the autonomous weapon systems well outside enemy airspace, allowing the weapon systems to seek out the target it was trained to destroy, and minimize the danger placed on the pilot.

1.2. Problem Statement

The thesis problem is to classify tactical targets as viewed from forward looking infrared (FLIR) imagery, and doppler imagery. The classifier to be used is a computer simulation of an ANN.

1.3. Scope

The targets of interest to be classified were from a tactical scenario. Doppler and FLIR imagery in raw form, must be preprocessed before being submitted as the input to an ANN. Much of this preprocessing is beyond the scope of this thesis effort; however, when deemed necessary the reader will be directed to the applicable reference. The targets to be classified from the doppler imagery consisted of M60 tanks, Petroleum, Oil and Lubricant (POL) tankers, jeeps, and 2.5 ton trucks [12]. Targets extracted from the FLIR imagery consisted of M551 tanks, 2.5 ton flatbed trucks, M113 Armored Personnel Carriers (APCs), and CJ-5 jeeps [10].

The ANN architecture used for this study was the multilayer perceptron described by Richard P. Lippmann [4:15-18]. Back propagation techniques will be used for updating the network weights. The minimization algorithms used were first and second order backward error propagation methods. The second order algorithm is a generalized version of the first order algorithm and is also an approximation to Newton's method, derived by David B. Parker [7:593-600; 8].

1.4. Approach and Methodology

The second order back propagation network required validation before being tested and used as a classifier. Therefore, before addressing the pattern classification problem, the network will be tested on the exclusive OR (XOR) problem. If the network can solve the XOR problem, then it may be possible to apply the network on the more difficult task of pattern classification.

Next, if the potential exists for pattern classification, it would be helpful to have a training set of feature vectors which have already been classified with a neural network. Such is the case with the doppler imagery. Dennis Ruck [12] trained a network using an algorithm provided by Richard P. Lippmann [4:17], and using moment invariants extracted from the doppler imagery. The algorithm was a first order, steepest decent search technique applying a momentum term. An important result of Ruck's study, for this thesis effort, was that the network achieved near perfect classification of the training set. Therefore, the doppler imagery will play an important role during the network validation stage.

A comparison between the first order and second order techniques using this data will follow. Classification accuracy of the training set and the test data set will be measured against number of iterations. Moment invariants, from the same imagery as the training set and never before seen by the network will make up the test data set. Also, log error plots versus the log of the number of iterations will be generated for comparison.

The final task was classification of features generated from the FLIR imagery. Various other features, as well as the moment invariants generated from the FLIR imagery will be considered for classification. A portion of the features will be used for classification and comparison with a Bayesian classifier implemented by Mike Roggemann [10]. The task will consist of training the network with a known training set and measuring the classification accuracy once the network has been trained. Again, classification accuracy will be measured using the training set and test set. First and second order methods will be used in comparison with the Bayesian classifier.

To conclude the section on classification of the FLIR imagery targets, the moment invariants will be considered explicitly for classification. Similar comparisons will be drawn as described above for the doppler imagery, for both the first and second order techniques.

1.5. Thesis Organization

This chapter served as the introduction to the thesis effort undertaken. Chapter two provides a discussion of the fundamental foundation of material necessary for the origins of the algorithms developed in chapter three. Chapter four consists of the validation stage for the second order back propagation model. Chapter five contains the results obtained during classification of the features generated from the FLIR imagery. Conclusions, recommendations, and discussions of the results follow in chapter six to conclude the thesis.

2. Background Material

2.1. Introduction

This chapter begins with a brief and limited discussion on preprocessing the doppler imagery, and forward looking infrared (FLIR) imagery. Next, an introduction to artificial neural networks (ANNs) follows, along with a brief discussion on the current use of ANNs as classifiers. The network architecture, learning algorithm, and minimization algorithms used for this study will be included. The following section highlights the significant steps in Parker's approximation to Newton's method [8], which in turn will be followed by the equations used to implement this approximation. The final section includes a discussion of the Bayesian classifier.

This investigation required a review of the methods and approximations used in solving differential equations and their discrete counterparts, the difference equation. Therefore appendix A has been reserved for a review in these areas. The algorithms involved are also quite heavily dependent on linear algebraic forms, so appendix B has been reserved for such discussions, along with any accompanying notation.

2.2. Image Preprocessing

2.2.1. Moment Invariant Feature Vectors

As mentioned in section 1.3, objects of interest, the targets, must be preprocessed before being applied to an ANN. The targets must be extracted from the raw doppler and FLIR imagery. This process of extraction is known as segmentation. Dennis Ruck [12] describes the segmentation of the doppler imagery. Where as Mike Roggemann [10] used a variation of the techniques described by Azriel Rosenfeld [11:62-73] for segmenting the objects from the FLIR imagery.

Once the targets have been segmented, a set of features is described to provide shape discrimination between targets. Both sets of images used moment invariants for shape description. Ruck [12] describes the technique for shaping the doppler imagery, while Roggemann [10] used a technique described by Ming-Kuei Hu [2:179-187].

The final preprocessing concerns normalizing the moment invariants. It was required to normalize the moments to insure that the large valued moments did not bias the decision making ability of the classifier. Very large values could influence the network in the wrong direction. Therefore both data sets were normalized to have a mean vector of zero and a standard deviation vector of one. This was accomplished by first computing the mean and standard deviation of each feature over the entire training set [1:99-105]. The j^{th} component of each feature vector in the training set, x_j, was transformed by the following equation:

$$y_j = \frac{x_j - m_j}{\sigma_j}, \qquad 2.1$$

where m_j and σ_j are the mean and standard deviation of feature j, respectively [1:99-105]. This provides a mean vector of 0 and a standard deviation vector of 1, and each feature is now scaled identically [12].

As eluded to in the previous paragraph, the normalized moment invariants are basically a set of features describing the target of interest. Therefore, an n-dimensional vector or feature vector describes and discriminates the targets of interest, where n represents the number of moment invariants. From here on these moment invariants will be referred to as feature vectors, representing the targets of interest. Each target, known as a class, will have many examples of feature vectors describing it.

2.2.2. Other Features

There were other features worthy of classification within the Roggemann FLIR imagery data set [10]. Roggemann performed classification with a decision rule of target (TGT) or non-target (NT) using his implementation of a Bayesian classifier. An identical classification will be performed with the ANN classifier studied in this effort for comparison. Target features extracted from the imagery considered tanks, trucks, APCs, and CJ-5 jeeps. Each image of a tactical scenario consisted of TGT blobs and NT blobs. Features extracted from these blobs and considered for this study were the length to width ratio of each blob, the blob mean intensity minus the background mean intensity, and the blob standard deviation of the intensity.

2.3. Introduction to Artificial Neural Networks

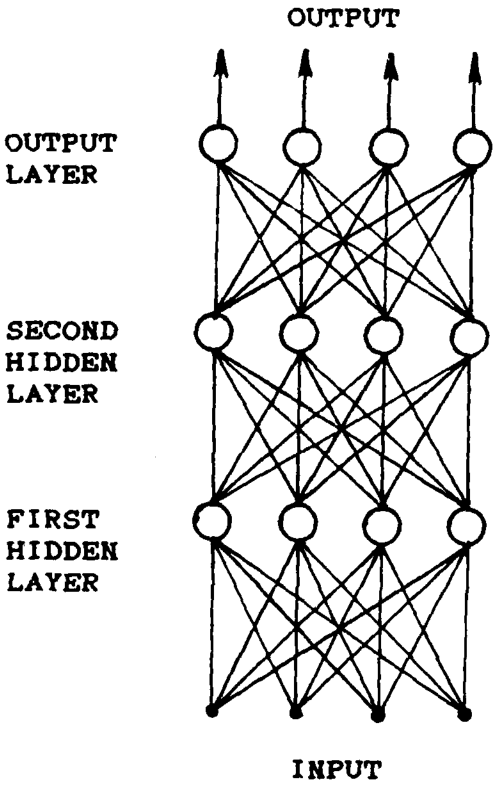
An ANN *is basically* a *computing system* usually consisting of many processing elements densely interconnected via interconnection weights. These processing elements are commonly referred to as nodes or cells. The construction of the network, the way in which the nodes are connected, is known as the network architecture.

Many architectures exist in the literature and each is highly dependent on the application [4:4-22]. In this study the ANN will be used as a classifier. In general, classifiers can perform three different tasks, as described by Lippmann [4:6]. First, they can identify which class best represents an input pattern, when the input has been corrupted by noise. Secondly, they can be used as an associative content-addressable memory. In this application, part of an input is available and the complete input pattern is desired. Such an application could be found in the decoding of information signals. The third task involves vector quantization. The idea is to map an n-dimensional input vector into an m-dimensional output vector, where usually $m < n$.

This thesis effort involves the identification of a class which best represents an input. The feature vectors discussed above will be used as the input patterns fed to the ANN. The ANN used in this study will project an n-dimensional feature vector to an m-dimensional output vector. This should not be confused with vector quantization. The resultant output describes the predetermined class from where the input originated. Hence, the term "supervised" network. The network architecture used to perform this task is discussed in the following section.

2.3.1. Multilayer Perceptron

A common architecture used for pattern classification applications is the multilayer perceptron [4:15-18], see Fig. 2.1. The multilayer perceptron consists of one or more hidden layers, where each node of each layer is connected to each node in the layer above it. This implies that each node is a multi-input, single-output element.



$$f^3_{out,j} = f\left(\sum_i^{N_2-1} w^3_{ij} \cdot f^2_{out,i} + \theta^3_j \right) \qquad 0 \le j \le N_3-1$$

$$f^2_{out,j} = f\left(\sum_i^{N_1-1} w^2_{ij} \cdot f^1_{out,i} + \theta^2_j \right) \qquad 0 \le j \le N_2-1$$

$$f^1_{out,j} = f\left(\sum_i^{N-1} w^1_{ij} \cdot f_{in,i} + \theta^1_j \right) \qquad 0 \le j \le N_1-1$$

Figure 2.1 Typical Three Layer Multilayer Perceptron Architecture

In Fig. 2.1, the numerical superscript notation denotes the parameter associated with its corresponding layer. The letter (i) denotes the number of inputs to an arbitrary cell, where j denotes the number of cells in a layer. In the following section, emphasis is place on notation for clarity.

2.3.2. Notation

Let $f_{out,j}(t)$ denote the output of the j^{th} node of a given layer in the network at time t. Furthermore, let $\mathbf{f}_{in}(t)$ denote the pattern of inputs to that node. Note that all bold face characters denote vectors. For example, $\mathbf{f}_{in}(t)$ is a vector whose components are

$$\mathbf{f}_{in}(t) = [\; f_{in,1}(t) \quad f_{in,2}(t) \quad \ldots \quad f_{in,q}(t) \;]^T.$$

Inputs are either the outputs of nodes from the previous layer or information from the environment, as shown in Fig. 2.1. The interconnection weight w_{ij} connects the output of the i^{th} node in the previous layer to the j^{th} node of the following layer. Therefore, $\mathbf{w}$ is the weight matrix for a given layer.

All though not shown in Fig. 2.1, each node will also receive a number of error signals back propagating from the layer immediately above it. These error signals make up a vector denoted as,

$$\mathbf{e}_{in}(t) = [\; e_{in,1}(t) \quad e_{in,2}(t) \quad \ldots \quad e_{in,r}(t) \;]^T.$$

Just as a node receives a number of weighted inputs to produce an output, the node will receive a number of weighted error signals.

The weighted error signals are summed by each node and this total error (e_{tot}) is used for updating the nodal weights and back propagating to the lower layers. The total error is

$$e_{tot}(t) = \sum_{i=1}^{r} e_{in,i}(t).$$

The output error of the j^{th} node is denoted as $e_{out,j}(t)$ and is defined below in section 2.4.3 and discussed in chapter three.

The algorithm used in this study requires the use of time derivatives of the above quantities. Therefore primed (′) quantities will denote the time derivative.

2.3.3. Backward Error Propagation

Within the confines of this thesis, backward error propagation (BEP) or backprop will be implied as an entire supervised learning algorithm [16:265]. This algorithm will be defined with a sigmoidal transfer function, a square error function, and a weight update rule to be defined in section 2.4.

The multilayer perceptron in Fig. 2.1 is an example of a backprop network, as introduced by Lippmann [4:15-18]. Input signals enter the bottom of the network and exit the top as output signals. The output signals are computed as functions of the inputs to the node and interconnecting weights. From this output, an error signal is computed and re-enters the top of the network propagating backwards. Hence, the name backprop. Each node within a given layer contains a set of weights that the cell must adjust to minimize the error signals.

Lippmann's first order minimization method uses the backprop learning algorithm. This algorithm computes the partial derivatives of the square error function with respect to the weights of the network. It uses these partial derivatives to update the weights.

When computing an output, each node within the network described by Lippmann performs two functions [4:17], as shown in Fig. 2.2.

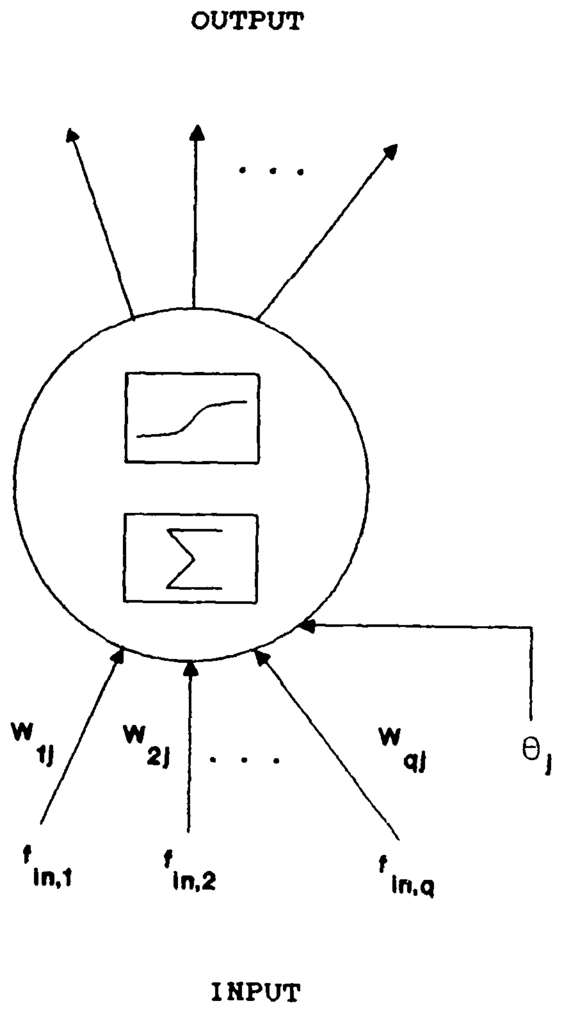


Figure 2. 2 Functions of a Single Cell on Forward Pass

First, it computes a weighted sum of all its inputs, the activation

$$\alpha_j(t) = \sum_i w_{ij}(t) \cdot f_{in,i}(t) + \theta_j. \qquad 2.2$$

The symbol θ_j is the threshold level of the j^{th} node. The threshold is no more than a weight, with a corresponding constant input normally equal to 1. Thus the name threshold, which is sometimes referred to as an offset. Secondly, it passes this weighted sum through a sigmoidal transfer function, where the output of the nonlinearity is the output of the node. The nonlinearity most commonly used for problems associated with the multilayer perceptron is a sigmoid function,

$$f_{out,j}(\alpha_j) = \frac{1}{(1 + \exp(-\alpha_j))}. \qquad 2.3$$

As shown in Fig. 2.3, the output of each node is continuous between 0 and 1. Parker [8] refers to this process as the forward pass.

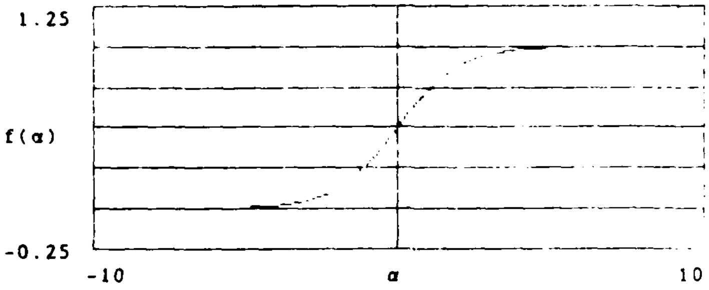


Figure 2.3 Sigmoid Transfer Function

2.3.4. Error Signals

Once the output of the network is determined, an error signal is computed to measure the performance of the network. The error signal is back propagated to all the layers. The weights of each node are adjusted using this error signal that it receives from all of the nodes which receive its output. Parker [8] refers to this process as the backward pass and Fig. 2.4 depicts the function of the cell on the backward pass.

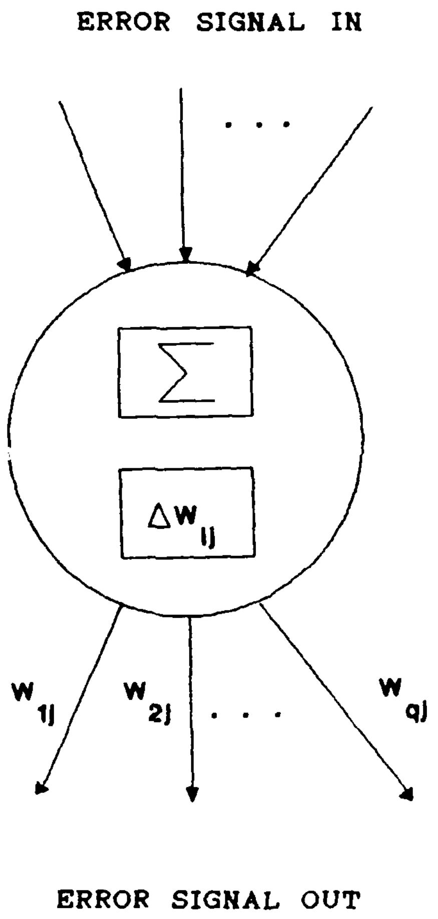


Figure 2.4 Functions of Single Cell on Backward Pass

In the following paragraphs a minimization algorithm or weight update rule is provided. The algorithm describes the method in which the node uses the error signals to update its weights, as described by Lippmann [4:17].

2.3.5. First Order Minimization

Lippmann's first order minimization technique assumes a squared error function to minimize [4:17]. The weight update rule uses the first partial of the squared error function with respect the weights of each cell. Performing this partial yields the following weight update rule for an arbitrary output layer node:

$$w_{ij}(t+1) = w_{ij}(t) + \cap \cdot \partial_j \cdot f_{in,\,i}(t) + \sigma \cdot (w_{ij}(t) - w_{ij}(t-1)).$$

The symbol $\cap$ controls the rate of convergence, while σ is a momentum scalar. The error signal ∂_j for the j^{th} output node has the following form:

$$\partial_j = f_{out,\,j} \cdot (1 - f_{out,\,j}) \cdot (d_j - f_{out,\,j})$$

where d_j denotes the desired output of the j^{th} output node. The desired value is commonly set to 1 or 0 and only one output node allowed high at a time. The error signal for the internal layer nodes is given by

$$\partial_j = f_{out,\,j} \cdot (1 - f_{out,\,j}) \cdot \sum_K \partial_K \cdot w_{jK} .$$

The error signal is a weighted summation over all nodes in the next higher layer. Keep in mind, that the output (f_{out}) in the above equation now pertains to the corresponding hidden layer cell.

The above update equations, follow the gradient of steepest descent in an iterative fashion. An input enters the bottom of the network and an output is computed at the top. The partial derivative of the squared error function is computed and back propagated as an error signal. This error signal is in turn used in updating the previous weight value of an arbitrary cell. This iterative process is continued until the minimum of the squared error function is found indicating optimum weight values.

The momentum term has the effect of smoothing the squared error surface. It provides more information on the current update cycle by adding a weighted change from the previous cycle. The momentum term pushes the change in weights further in the direction of the previous update. Appendix D considers the momentum term in more detail.

2.4. Introduction to Second Order Minimization Techniques

In order for classification to take place, there must be some learning rule the network uses to minimize the error associated with the decisions it must make. Therefore the learning rule applies some minimization technique. Dennis Ruck [12] implemented a multilayer perceptron with a backprop learning rule that applied the momentum method, as provided by Lippmann [4:17] and discussed in the previous section. The momentum method, developed by Rummelhart, Hinton and Williams [13], is a variation of the steepest decent method developed by Werbos [15]. Both methods are first order methods because they only involve the first derivative of the quantity being minimized. A second order algorithm makes use of the second derivatives.

To understand the difference between first and second order techniques, Parker draws upon a simple, but quite effective analogy which is quoted below.

> "Imagine that you are at the top of a ridge. Below you is a long, narrow valley that slopes gently down to your right. Far off in the valley to the right is the ski lodge, to which you wish to return. One way to get to the lodge is to simply sit on your skis and let gravity move you. You will zip quickly down the slope till you hit the valley, but once in the valley you will coast very, very slowly till you reach the lodge" [8].

This is equivalent to first order techniques which follow the gradient of steepest decent. Fast convergence down the slope may give way to very slow convergence in a valley. On the other hand consider an alternate path.

> "A better way to get back to the lodge is to slightly drag one of your skis so that you cut across the slope, maintaining a constant speed till you hit the lodge.

This is equivalent to a second order algorithm, which has a constant convergence rate under appropriate conditions" [8].

The algorithm discussed in the following paragraphs is an approximation developed by David Parker [7:593-600; 8] to the second order Newton's method. This algorithm is a more general case of the steepest decent and momentum methods. By adjusting the learning parameters correctly the algorithm can be made to perform as the steepest decent or momentum method. Below is a brief presentation of the algorithm, for a more thorough explanation see [8] and appendix A.

2.4.1. Performance Surface

For now, the quantity being minimized for this study is an independent variable of some performance function of the network. Parker denotes the instantaneous performance of a network by $s(\ \mathbf{f}_{in}(t),\ \mathbf{w}(t)\ )$. The instantaneous performance is dependent on the current set of inputs and also the current set of weights. However, in general, the derivation begins by defining an average instantaneous performance, which depends only on the weights of the network. The average instantaneous performance is given by,

$$\operatorname{avg}(\ s(\mathbf{w}(t))\ ) : \mu \cdot \int_{-\infty}^{t} s(\ \mathbf{f}_{in}(\tau),\ \mathbf{w}(t)\ ) \cdot e^{-\mu \cdot (t-\tau)}\, d\tau. \qquad 2.4$$

Parker notes that the scalar quantity μ is roughly the inverse of the amount of time the average is considered. Basically, the instantaneous performance will be exponentially weighted over all input patterns $f_{in}(\tau)$ for a set of weights $w(t)$ at time t. In other words, the average performance provides a certain amount of memory from information of past inputs. The exponential term insures that emphasis is placed on the most current inputs [8].

A graph of avg ($s(w)$) as a function of $w(t)$ would define a performance surface at a fixed time t. Thus, the performance surface changes over time with each new set of inputs. According to Parker, the task of the backprop network is to find the lowest point on the performance surface, and then follow that point as the surface changes with time [8].

2.4.2. Second Order Differential Equation

Parker's derivation of the second order differential equation is very thorough and well explained. Therefore, no attempt will be made to duplicate his work in full. However, it will be time well spent to highlight the significant intermediate equations, as well as the final result. See [8] and Appendix A for further study.

Parker derives the algorithm with an objective of optimality in mind. Assuming the weights have converged to a minimum of the performance surface, then as the performance surface changes with time, the weights should follow this minimum. The first step involves the derivation of Newton's method from an optimality criterion. The goal is to find the minimum of the performance surface by updating the weights. So the first step is to take the derivative of both sides of Eq. 2.4 with respect to the weights. Since the performance surface is changing with time, it is desired to have the weights follow the minimum as time changes. This requires taking the time derivative of both sides. By doing so Eq. 2.4 is transformed to the following (see appendix A):

$$\frac{\partial w^*}{\partial t} = -\left[\operatorname{avg}\left(\frac{\partial^2 s}{\partial w^* \partial w^{*T}} \right) \right]^{-1} \cdot \mu \cdot \frac{\partial s}{\partial w^*}, \qquad 2.5$$

where the functional dependencies on t, $f_{in}(\tau)$, and $w(t)$ have been suppressed for convenience. The star (*) notation denotes the optimal value of the weights (w^*). The following relationship can be made since the network performance is a function of time through the weights:

$$\frac{\partial}{\partial t}\left(\frac{\partial s}{\partial w^*} \right) = \left(\frac{\partial^2 s}{\partial w^* \partial w^{*T}} \right) \cdot \frac{\partial w^*}{\partial t}.$$

As Parker points out, the explicit first order differential equation of Eq. 2.5, known as Newton's method, is valid only if the average second derivative matrix is invertible (it is not) [7:593-600; 8]. By actually computing the determinant of the time average second derivative matrix, reveals the matrix to be singular and thus, not invertible. This is shown in appendix D. Regardless, inverting this matrix is entirely too expensive. Consider n weights in the network, the number of operations performed is a function of n^3 or $O(n^3)$. The reason behind the potentially enormous number of operations is that each component of the matrix must be computed. This entails computing the average second partial with respect to every combination of weights, followed by inverting the matrix for each cell in the network. These computations must be made before the weights are updated. A very unpleasant thought! This task has been avoided by other researchers using quasi-Newton methods reducing the number of operations to $O(n^2)$.

Parker, on the other hand chose a different route. Rewriting Eq. 2.5 as

$$\text{avg}\left(\frac{\partial^2 s}{\partial w^* \partial w^{*T}} \right) \cdot \frac{\partial w^*}{\partial t} = -\mu \cdot \frac{\partial s}{\partial w^*}, \qquad 2.6$$

an iterative approach is applied to obtain a close approximation to the time derivative of w^* [8]. Appendix A describes an iterative approach in general. Following this approach, a second order differential equation for an optimal path for the weights:

$$\frac{\partial^2 w^+}{\partial t^2} = -\beta \cdot \mu \cdot \frac{\partial s}{\partial w^+} - \beta \cdot \text{avg}\left(\frac{\partial^2 s}{\partial w^+ \partial w^{+T}} \right) \cdot \frac{\partial w^+}{\partial t}, \qquad 2.7$$

where β controls the convergence of the algorithm. The symbol w^+ denotes the approximate values of w^*. The derivation could be stopped here with an attempt to implement Eq. 2.7. However, Parker chooses to continue since Eq. 2.7 is only an approximation for the optimal path [8].

According to Parker, leakage terms are required to guarantee convergence [7:593-600; 8], see appendix A. The final version of the algorithm used for classification in this thesis is given by,

$$\frac{\partial^2 w}{\partial t^2} : -a_1 \cdot \frac{\partial s}{\partial w} - \left(a_2 \cdot I + a_3 \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \right) \cdot w$$

$$- \left(a_4 \cdot I + a_5 \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \right) \cdot \frac{\partial w}{\partial t}. \qquad 2.8$$

The matrix I is the identity matrix, and is introduced to ensure that matrices are added to matrices. Where the constants

$a_1 : \beta \cdot \mu$,

$a_2 : l_1 \cdot l_2$,

$a_3 : l_1 \cdot \beta$,

$a_4 : l_1 + l_2$, and

$a_5 : \beta$

are learning parameters and usually small positive numbers. The constants l_1 and l_2 are the leakage terms introduced by Parker [8], see appendix A.

2.4.3. Second Order Implementing Equations

Implementing the algorithm of Eq. 2.8 is not a straight forward exercise. Chapter three will derive the implementing equations in detail. There are several ways to implement the algorithm and the implementing equations Parker uses are listed below. The equations describe a forward sweep and backward sweep through the network. On the forward sweep each cell computes its own copy of the following:

$$f_{out,k} = f_{out,k}(f_{in,k}, w_k),$$

$$w'_k = a_3 \cdot \Delta t \cdot w_k + a_5 \cdot \Delta w_k,$$

$$f'_{out,k} = \left.\frac{\partial f_{out}}{\partial f_{in}^T}\right|_k \cdot f'_{in,k} + \left.\frac{\partial f_{out}}{\partial w^T}\right|_k \cdot w'_k .$$

The calculations for the backward sweep are as follows:

$$e_{tot} = \sum_{i=1}^{r} e_{in,i} = 1^T \cdot e_{in},$$

$$e'_{tot} = \sum_{i=1}^{r} e'_{in,i} = 1T \cdot e'_{in},$$

$$e_{out} = e_{tot} \cdot \left.\frac{\partial f_{out}}{\partial f_{in}}\right|_k,$$

$$e'_{out} = e'_{tot} \cdot \left.\frac{\partial f_{out}}{\partial f_{in}}\right|_k + e_{tot} \cdot \left(\left.\frac{\partial^2 f_{out}}{\partial f_{in} \partial w^T}\right|_k \cdot w' + \left.\frac{\partial^2 f_{out}}{\partial f_{in} \partial f_{in}^T}\right|_k \cdot f'_{in} \right),$$

$$\Delta w_{K+1} = \Delta w_K - a_2 \cdot \Delta t^2 \cdot w_K -- a_4 \cdot \Delta t \cdot \Delta w_K$$

$$+ \left(a_1 \cdot \Delta t^2 \cdot e_{tot} + \Delta t \cdot e'_{tot} \right) \cdot \left. \frac{\partial f_{out}}{\partial w} \right|_K$$

$$+ \Delta t \cdot e_{tot} \cdot \left(\left. \frac{\partial^2 f_{out}}{\partial w \partial w^T} \right|_K \cdot w' + \left. \frac{\partial^2 f_{out}}{\partial w \partial f_{in}^T} \right|_K \cdot f'_{in} \right),$$

$$w_{K+1} = w_K + \Delta w_{K+1}.$$

The above equations describe a discrete implementation of Eq. 2.7 for a single cell, where K denotes the discrete time step [8]. The implementation may be simulated with a computer program. They are listed here for those readers who wish to skip the detailed implementation stage discussed in chapter three. Figure 2.5 below demonstrates how the cell varies from Figs. 2.2 and 2.4 in the amount of information it must process.

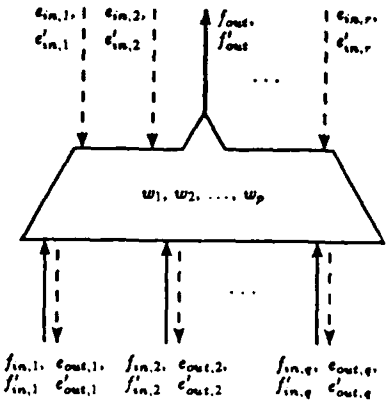

$e_{in,1}$, $e'_{in,1}$
$e_{in,2}$, $e'_{in,2}$
f_{out}, f'_{out}
$e_{in,r}$, $e'_{in,r}$
$w_1, w_2, \ldots, w_p$
$f_{in,1}$, $e_{out,1}$, $f'_{in,1}$ $e'_{out,1}$
$f_{in,2}$, $e_{out,2}$, $f'_{in,2}$ $e'_{out,2}$
$f_{in,q}$, $e_{out,q}$, $f'_{in,q}$ $e'_{out,q}$

Figure 2.5 Signal Flow Through Cell Using Second Order Implementation [8]

2.5. Bayesian Classifier

Later, in chapter five, the results obtained from a Bayesian classifier will be compared to the results obtained from an ANN classifier for a given set of features. In light of this, the text below provides an introduction to the concept of a Bayesian classifier. In particular, the approach Roggemann used in his implementation of a Bayesian classifier will be discussed [10]. The discussion begins with a statement of Bayes rule, followed by its application in the Bayes classifier.

2.5.1. Implementation

Recall that Bayes rule is defined in the following way:

$$p[A/B] = \frac{p[A,\ B]}{p[B]}$$

and

$$p[B/A] = \frac{p[A,\ B]}{p[A]}$$

such that

$$p[B/A] = \frac{p[A/B] \cdot p[B]}{p[A]}.$$

The probability of the occurrence of B given A (p[B/A]) is equal to the product of the probability of A given B (p[A/B]) and the probability of B (p[B]), divided by the probability of A (p[A]).

For the Bayes classifier, the idea is to determine the probability of the occurrence of a target (TGT) given some feature (F) describing the target, or p[TGT/F]. Another decision the *classifier* must make is to determine the probability of a non-target (NT) given a feature, or p[NT/F]. Each of these conditionals is determined from the probability of occurrence of a feature given a TGT or a NT. Hence, the known conditionals exhibited are in the form of p[F/TGT] and p[F/NT]. Thus, Roggemann's implementation considered classifications of target (TGT) and non-target (NT) only. In other words, by applying Bayes rule it is desired to compute the following:

$$p[TGT/F] = \frac{p[F/TGT] \cdot p[TGT]}{p[F]}$$

and

$$p[NT/F] = \frac{p[F/NT] \cdot p[NT]}{p[F]}.$$

The value of the probability of a feature is given as:

$$p[F] = p[F/TGT] \cdot p[TGT] + p[F/NT] \cdot p[NT].$$

Applying the "Principle of Indifference" the a priori probabilities of p[TGT] and p[NT] are equal to 0.5 [5:1-53].

To consider multiple features (F_1, F_2, ..., F_n), it's necessary to impose a conditional dependence on the conditionals for multiple feature decisions, such that

$$p[F_1, F_2, \ldots, F_n/TGT] = p[F_1/TGT] \cdot p[F_2/TGT] \cdot \ldots \cdot p[F_n/TGT]$$

$$= \prod_{i=1}^{n} p[F_i/TGT]$$

Applying Bayes rule while considering the above alterations, provides:

$$p[TGT/F_1, F_2, \ldots, F_n] = \frac{\pi\ p[F_i/TGT] \cdot p[TGT]}{\pi\ p[F_i/TGT] \cdot p[TGT] + \pi\ p[F_i/NT] \cdot p[NT]}$$

and

$$p[NT/F_1, F_2, \ldots, F_n] = \frac{\pi\ p[F_i/NT] \cdot p[NT]}{\pi\ p[F_i/TGT] \cdot p[TGT] + \pi\ p[F_i/NT] \cdot p[NT]}$$

where π is understood to range over all features, $i = 1 .. n$.

Now that the desired a posteriori conditionals have been defined, it's necessary to define a decision criterion. The criterion used by Roggemann [10] is the maximum a posteriori (MAP) decision criterion approximating the minimum probability of error. Simply choose TGT, if the $p[TGT/F] > p[NT/F]$, otherwise select NT [5:1-53].

2.5.2. Addition of Bins

The conditional probability distribution function (PDF) of an arbitrary feature given a TGT (or NT) is in general a continuous function. Therefore, the probability of a feature lying on a single feature point given a TGT (or NT) is zero. For instance,

$$p[F = f_0/TGT] = 0.$$

Therefore, the conditional PDF is broken up into several uniform incremental regions, called bins. Hence, the conditional probability actually desired is given by

$$p[f_0/TGT \le F < f_1/TGT].$$

The number of bins varies and allows the construction of a new discrete PDF as a function of the number of bins. Figure 2.6 displays a typical conditional PDF (p[F:OBJECT TRUTH]) as a function of the number of feature bins. The (+) notation is the conditional PDF for TGT data, while (o) indicates NT data, where OBJECT TRUTH represents TGT or NT.

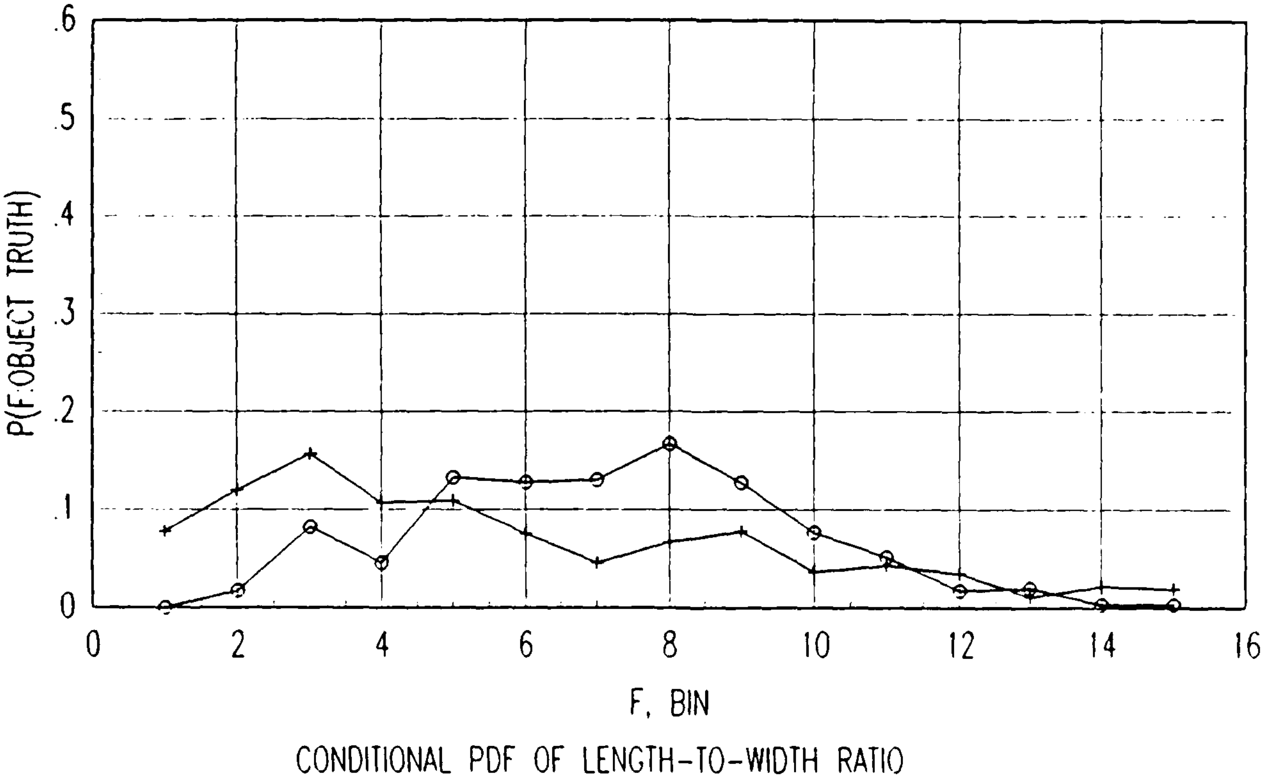


Figure 2.6 Typical Conditional PDF

2.6. Summary

This chapter focused on the background material necessary for understanding the thesis effort undertaken. The chapter began with a discussion of imagery preprocessing. It was found that the images were reduced to a set of features describing and discriminating the targets of interest. These features represent the final interpretation of the real world. Next, it was desired to introduce the ANN classifier used in this study, the multilayer perceptron. The network required a learning algorithm and backprop was chosen using first and second order minimization techniques. Following a discussion on the performance surface used for this study, the second order derivation introduced by David Parker was highlighted. Appendix A discusses the derivation in more detail. Equation 2.8 represents the second order approximation to Newton's method. Appendix D discusses convergence considerations for a true second order Newton's method. It is in appendix D, where it is reasoned that Eq. 2.5 must be approximated. Chapter two concluded with a brief introduction to the Roggemann implementation of a Bayesian classifier.

The following chapter, discusses in detail the implementation of the differential equation provided by Eq. 2.8. The approximations assumed, as well as the efforts to reduce the computational overhead will be covered. The latter part of chapter three will consider the initial network setup.

3. Second Order Algorithm and Network Convergence

3.1. Introduction

The first five sections of this chapter are devoted to the implementation stages of Parker's second order derivation introduced in chapter two. Specifically, the implementation of Eq. 2.8 will be discussed. The following sections describe in detail the approximations made, along with a discussion on the mathematical notation. In addition, the implementing equations of section 2.4.3 will be explained in full.

Once the algorithm used in this study has been fully defined, a discussion on the initial state of the network is necessary. For example, what are the initial network parameter values? How are the parameters chosen? These questions will be addressed in section 3.6.

3.2. Definition of Network Performance

For the problem at hand, the parameter to be minimized will be defined to be the squared error function. Where the instantaneous performance for a single output node is defined to be

$$\begin{aligned} s(f_{in}(t),\ w(t)) &:= (\ d_{out}(t) - f_{out}(f_{in}(t),\ w(t))\ )^2 \\ &:= (\ e(t)\ )^2. \end{aligned}$$

The performance surface is defined to be a function of the desired output, $d_{out}(t)$, and the actual output at time t, and will be interpreted as the square of the error e(t).

As per Eq. 2.2, the output has the following form,

$$f_{out}(f_{in}(t),\ w(t)) = \frac{1}{(1 + \exp(-f_{in}^{T} \cdot w + \Theta))}. \qquad 3.1$$

In this context, w pertains to just those weights associated with the cell in question.

Since there can be many output nodes, there must be an expression for s to accommodate the vector of error signals generated from the output layer. Therefore, in general s may be rewritten as

$$s = e^{T} \cdot e \qquad 3.2$$

The dependencies of s on $f_{in}(t)$ and w(t), and e(t) on t has been suppressed for notational convenience.

3.2.1. First Partial Derivative

Now that a quantity of performance has been defined, the first and second partial derivatives of the performance indicator (s) need to be defined. The first partial derivative of s with respect to the weights is defined as follows, •

$$\frac{\partial s}{\partial w} = \frac{\partial}{\partial w}(e^{T} \cdot e)$$

$$= 2 \cdot \frac{\partial e^{T}}{\partial w} \cdot e, \qquad 3.3$$

since

$$\frac{\partial \mathbf{e}^T}{\partial \mathbf{w}} \cdot \mathbf{e} = \mathbf{e}^T \cdot \frac{\partial \mathbf{e}}{\partial \mathbf{w}}.$$

The chain rule was used to obtain Eq. 3.3. The partial derivative of the transpose of **e** with respect to **w** is a matrix and is defined in appendix B. In this context, **w** implies all the weights of the network.

3.2.2. Second Partial Derivative

The second partial of s with respect to the weights is found by applying the chain rule once again. Therefore,

$$\begin{aligned}
\frac{\partial^2 s}{\partial \mathbf{w} \partial \mathbf{w}^T} &= \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}^T}(\mathbf{e}^T \cdot \mathbf{e}) \\
&= \frac{\partial}{\partial \mathbf{w}} \cdot \left(\frac{\partial}{\partial \mathbf{w}^T}(\mathbf{e}^T \cdot \mathbf{e}) \right) \\
&= \frac{\partial}{\partial \mathbf{w}} \cdot \left(2 \cdot \mathbf{e}^T \cdot \frac{\partial \mathbf{e}}{\partial \mathbf{w}^T} \right) \\
&= 2 \cdot \left(\frac{\partial \mathbf{e}^T}{\partial \mathbf{w}} \cdot \frac{\partial \mathbf{e}}{\partial \mathbf{w}^T} + \mathbf{e}^T \cdot \frac{\partial^2 \mathbf{e}}{\partial \mathbf{w} \partial \mathbf{w}^T} \right)
\end{aligned}$$

$$\frac{\partial^2 s}{\partial \mathbf{w} \partial \mathbf{w}^T} = 2 \cdot \left(\frac{\partial \mathbf{e}^T}{\partial \mathbf{w}} \cdot \frac{\partial \mathbf{e}}{\partial \mathbf{w}^T} + \frac{\partial^2 \mathbf{e}^T}{\partial \mathbf{w} \partial \mathbf{w}^T} \cdot \mathbf{e} \right) \qquad 3.4$$

where the partials of **e** and $\mathbf{e}^T$ with respect to $\mathbf{w}^T$ and **w** respectively, and the second partial of $\mathbf{e}^T$ with respect to **w** and $\mathbf{w}^T$ are defined in appendix B. Again, in the context above, **w** is a vector of weights containing all the weights in the network.

3.2.3. Approximate Average Second Partial Derivative

Parker notes that implementing Eq. 2.8 explicitly requires $O(n^2)$ operations [8]. In order to reduce the number of operations the network would have to perform to update the weights, some approximations are in order. The first approximation concerns the average second partial derivative of the network performance quantity, s. Basically, to define the average second partial of the network performance,

$$\operatorname{avg}\left(\frac{\partial^2 s}{\partial w \partial w^T}\right)$$

would require an iterative approach to obtain a solution to this average second derivative matrix. The elements of the matrix depend on the behavior of the cell at some point in the past, considering the current weight values. Therefore, they cannot be computed without going back in time [7:598]. For some applications, including the application of this effort, this would require large storage space in the form of memory. However, Parker suggests an alternative for the class of semilinear functions [8]. The assumption is to approximate the average of the second partial of the network performance with the current instantaneous value, such that

$$\frac{\partial^2 s}{\partial w \partial w^T} \approx \operatorname{avg}\left(\frac{\partial^2 s}{\partial w \partial w^T}\right). \qquad 3.5$$

Since the network cells have semilinear transfer functions in the sigmoid function [16:264], according to Parker this should be a good estimate [8].

As it turns out, this approximation plays a significant role in arriving at the implementing equation Δw_{K+1} of section 3.3. If you recall, avg(s) was a function of $f_{in}(\tau)$ and $w(t)$. The only dependence of s on time was through $w(t)$. Now that the instantaneous value is desired the network performance is a function of $f_{in}(t)$ and $w(t)$, where both are functions of time.

The above approximation is also a desired result. Recall from Eq. 2.3 that the average network performance applied more weight to the most current input. The input data set used for this study was a set of feature vectors describing the target of interest. There is no reason to believe at this point, that any one of these feature vectors is more important than the others. Therefore, this assumption is considered a good estimate.

3.3. Algorithm Development

The development begins with a restatement of Eq. 2.8, where

$$\frac{\partial^2 w}{\partial t^2} = -a_1 \cdot \frac{\partial s}{\partial w} - \left(a_2 \cdot I + a_3 \cdot avg\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \right) \cdot w$$

$$- \left(a_4 \cdot I + a_5 \cdot avg\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \right) \cdot \frac{\partial w}{\partial t}. \qquad 3.6$$

Using the approximation of Eq. 3.5, Eq. 3.6 becomes,

$$\frac{\partial^2 w}{\partial t^2} = -a_1 \cdot \frac{\partial s}{\partial w} - \left(a_2 \cdot I + a_3 \cdot \frac{\partial^2 s}{\partial w \partial w^T} \right) \cdot w$$

$$- \left(a_4 \cdot I + a_5 \cdot \frac{\partial^2 s}{\partial w \partial w^T} \right) \cdot \frac{\partial w}{\partial t}. \qquad 3.7$$

As pointed out in section 2.4.1, Parker reasoned that since the average network performance was a function of t, because the weights were a function of t, that he could make the following relationship:

$$\frac{\partial}{\partial t}\left(\frac{\partial s}{\partial w} \right) = \left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \cdot \frac{\partial w}{\partial t} \qquad 3.8$$

For Eq. 3.7 to be totally correct, this action must be reversed since it is clear that the instantaneous second partial of s is a function of $f_{in}(t)$ and w(t). Therefore, substituting Eq. 3.8 in Eq. 3.7, and removing the identity matrix I, and then expanding, the following equation is obtained:

$$\frac{\partial^2 w}{\partial t^2} = -a_1 \cdot \frac{\partial s}{\partial w} - a_2 \cdot w - a_3 \cdot \frac{\partial^2 s}{\partial w \partial w^T} \cdot w$$

$$- a_4 \cdot \frac{\partial w}{\partial t} - a_5 \cdot \frac{\partial}{\partial t}\left(\frac{\partial s}{\partial w} \right). \qquad 3.9$$

The next step is to substitute Eqs. 3.3 and 3.4 into Eq. 3.9, such that

$$\frac{\partial^2 w}{\partial t^2} = -2\cdot a_1\cdot\frac{\partial e^T}{\partial w}\cdot e - a_2\cdot w - a_4\cdot\frac{\partial w}{\partial t}$$
$$- 2\cdot a_3\cdot\left(\frac{\partial e^T}{\partial w}\cdot\frac{\partial e}{\partial w^T} + \frac{\partial^2 eT}{\partial w\partial w^T}\cdot e\right)\cdot w$$
$$- 2\cdot a_5\cdot\frac{\partial}{\partial t}\left(\frac{\partial e^T}{\partial w}\cdot e\right)$$

and by applying the chain rule,

$$\frac{\partial}{\partial t}\left(\frac{\partial e^T}{\partial w}\cdot e\right) = \frac{\partial}{\partial w}\left(\frac{\partial e^T}{\partial t}\right)\cdot e + \frac{\partial e^T}{\partial w}\cdot\frac{\partial e}{\partial t}$$

and by regrouping like terms, Eq. 3.9 becomes,

$$\frac{\partial^2 w}{\partial t^2} = -2\cdot a_1\cdot\frac{\partial e^T}{\partial w}\cdot e - a_2\cdot w - a_4\cdot\frac{\partial w}{\partial t}$$
$$- 2\cdot\frac{\partial e^T}{\partial w}\cdot\left(a_3\cdot\frac{\partial e}{\partial w^T}\cdot w + a_5\cdot\frac{\partial e}{\partial t}\right)$$
$$- 2\cdot\left(a_3\cdot\frac{\partial^2 e^T}{\partial w\partial w^T}\cdot w + a_5\cdot\frac{\partial}{\partial w}\left(\frac{\partial e^T}{\partial t}\right)\right)\cdot e. \qquad 3.10$$

Recall that the error was defined as a vector, since there may be several error signals to back propagate. This implies that in the hidden layers, individual cells will be responsible for summing the input error signals and using this sum for updating the cell's weights. To clean up Eq. 3.10, the following quantities are defined:

$$e_{in} = 2 \cdot e$$

and

$$e_{tot} = \sum_{i=1}^{r} e_{in,\,i} = 1^{T} \cdot e_{in} \qquad 3.11$$

The upper bound, r, is the total number of error signals propagated to an individual cell. For instance, the upper bound on the total error (e_{tot}) propagated to the output layer cells is r = 1. The number of error signals propagated to the hidden layer cells depends on the number of cells in layer immediately above it. The upper bound will vary from layer to layer and remain constant within a layer. Rewriting Eq. 3.10 with Eq. 3.11 in mind, yields:

$$\begin{aligned} \frac{\partial^2 w}{\partial t^2} &= -a_1 \cdot e_{tot} \cdot \frac{\partial e^T}{\partial w} - a_2 \cdot w - a_4 \cdot \frac{\partial w}{\partial t} \\ &\quad - 2 \cdot \frac{\partial e^T}{\partial w} \cdot \left(a_3 \cdot \frac{\partial e}{\partial w^T} \cdot w + a_5 \cdot \frac{\partial e}{\partial t} \right) \\ &\quad - e_{tot} \cdot \left(a_3 \cdot \frac{\partial^2 e^T}{\partial w \partial w^T} \cdot w + a_5 \cdot \frac{\partial}{\partial w} \left(\frac{\partial e^T}{\partial t} \right) \right). \qquad 3.12 \end{aligned}$$

The time derivatives of e are now considered. From Eq. 3.2 it is clear that the error is a function of the desired and actual outputs. For the cases studied in this thesis effort, the desired output is considered to be piece-wise continuous and constant over the time in question. For any given input vector, each of the output desired responses is either 0 or 1. Thus, its time derivative is zero and the time derivative of the error signal for any single output cell in the network is

$$\frac{\partial e}{\partial t} = -\frac{\partial f_{out}}{\partial t}.$$

Since f_{out} is time dependent through $f_{in}(t)$ and $\mathbf{w}(t)$, the time derivative may be expressed as a function of these time dependent quantities. Therefore the time derivative of the error signal of a single cell becomes,

$$\frac{\partial e}{\partial t} = -\left(\frac{\partial f_{out}}{\partial \mathbf{w}^T} \cdot \frac{\partial \mathbf{w}}{\partial t} + \frac{\partial f_{out}}{\partial \mathbf{f}_{in}^T} \cdot \frac{\partial \mathbf{f}_{in}}{\partial t} \right). \qquad 3.13$$

Also note that

$$\frac{\partial e}{\partial \mathbf{w}^T} = -\frac{\partial f_{out}}{\partial \mathbf{w}^T} \qquad 3.14$$

and similarly

$$\frac{\partial e}{\partial \mathbf{w}} = -\frac{\partial f_{out}}{\partial \mathbf{w}} \qquad 3.15$$

for a single cell. Substituting Eqs. 3.13, 3.14 and 3.15 into 3.12, the weight update rule for a single cell becomes:

$$\frac{\partial^2 w}{\partial t} = a_1 \cdot e_{tot} \cdot \frac{\partial f_{out}}{\partial w} - a_2 \cdot w - a_4 \cdot \frac{\partial w}{\partial t} \qquad 3.16$$

$$- 2 \cdot \frac{\partial f_{out}}{\partial w} \cdot \left(a_3 \cdot \frac{\partial f_{out}}{\partial w^T} \cdot w + a_5 \cdot \left(\frac{\partial f_{out}}{\partial w^T} \cdot \frac{\partial w}{\partial t} + \frac{\partial f_{out}}{\partial f_{in}^T} \cdot \frac{\partial f_{in}}{\partial t} \right) \right)$$

$$+ e_{tot} \cdot \left(a_3 \cdot \frac{\partial^2 f_{out}}{\partial w \partial w^T} \cdot w + a_5 \cdot \frac{\partial}{\partial w} \left(\frac{\partial f_{out}}{\partial w^T} \cdot \frac{\partial w}{\partial t} + \frac{\partial f_{out}}{\partial f_{in}^T} \cdot \frac{\partial f_{in}}{\partial t} \right) \right).$$

Differentiation of the sigmoid function is a straight forward exercise and will not be covered here. Appendix C describes the various derivatives of the sigmoid in detail. Regrouping like terms in Eq. 3.16 results in

$$\frac{\partial^2 w}{\partial t^2} = e_{tot} \cdot \frac{\partial f_{out}}{\partial w} - a_2 \cdot w - a_4 \cdot \frac{\partial w}{\partial t}$$

$$- 2 \cdot \frac{\partial f_{out}}{\partial w} \cdot \left(\frac{\partial f_{out}}{\partial w^T} \cdot \left(a_3 \cdot w + a_5 \cdot \frac{\partial w}{\partial t} \right) + a_5 \cdot \frac{\partial f_{out}}{\partial f_{in}^T} \cdot \frac{\partial f_{in}}{\partial t} \right)$$

$$+ e_{tot} \cdot \left(\frac{\partial^2 f_{out}}{\partial w \partial w^T} \cdot \left(a_3 \cdot w + a_5 \cdot \frac{\partial w}{\partial t} \right) + a_5 \cdot \frac{\partial^2 f_{out}}{\partial w \partial f_{in}^T} \cdot \frac{\partial f_{in}}{\partial t} \right). \qquad 3.17$$

Since the artificial neural network (ANN) for this study will be run as a computer simulation, discrete mathematics are introduced. Difference equations, in lieu of differential equations are required. The following set of approximations are required for the transformation:

$$\frac{\partial^2 w}{\partial t^2} \approx \frac{\Delta^2 w_{K+1}}{\Delta t^2} = \frac{\Delta w_{K+1} - \Delta w_K}{\Delta t^2},$$

$$\frac{\partial w}{\partial t} \approx \frac{\Delta w_K}{\Delta t},$$

$$\Delta w_K = w_K - w_{K-1},$$

$$w_K' = a_3 \cdot \Delta t \cdot w_K + a_5 \cdot \Delta w_K,$$

$$\frac{\Delta f_{in}}{\Delta t} \approx \frac{\partial f_{in}}{\partial t},$$

$$f_{in,K}' = a_5 \cdot \Delta f_{in,K},$$

$\Delta t \equiv$ the amount of time occurring between time step K and K + 1.

The quantities w_K' and $f_{in,K}'$ can be thought of as average values, approximating the discrete time derivatives of w_K and $f_{in,K}$, respectively. Recall, that the error surface is changing instantaneously with each new input vector. The estimates of the time derivatives of w_K and $f_{in,K}$ provide the network with information of how the surface is changing with time. These time derivatives will no doubt be used in updating the cell weights.

With these approximations, apply the first two by substitutions and then multiply each side of Eq. 3.17 by Δt^2. Add Δw_K to both sides of the equation and then apply the approximate discrete time derivatives of w_K and $f_{in,K}$. The result is

$$
\begin{aligned}
\Delta w_{k+1} = {} & (1 - a_4 \cdot \Delta t) \cdot \Delta w_k - a_2 \cdot \Delta t^2 \cdot w_k + a_1 \cdot \Delta t^2 \cdot e_{tot} \cdot \left. \frac{\partial f_{out}}{\partial w} \right|_k \\
& - 2 \cdot \Delta t \cdot \left. \frac{\partial f_{out}}{\partial w} \right|_k \cdot \left(\left. \frac{\partial f_{out}}{\partial w^T} \right|_k \cdot w'_k + \left. \frac{\partial f_{out}}{\partial f^T_{in}} \right|_k \cdot f'_{in,k} \right) \\
& + \Delta t \cdot e_{tot} \cdot \left(\left. \frac{\partial^2 f_{out}}{\partial w \partial w^T} \right|_k \cdot w'_k + \left. \frac{\partial^2 f_{out}}{\partial w \partial f^T_{in}} \right|_k \cdot f'_{in,k} \right) .
\end{aligned}
\qquad 3.18
$$

The final approximation to the implementation stage concerns the time derivative of e_{tot}. Since

$$
\frac{\partial e}{\partial t} = - \frac{\partial f_{out}}{\partial t}
$$

for a single output cell, then e'_{tot} may be approximated by

$$
e'_{tot} = -2 \cdot \left(\left. \frac{\partial f_{out}}{\partial w^T} \right|_k \cdot w'_k + \left. \frac{\partial f_{out}}{\partial f^T_{in}} \right|_k \cdot f'_{in,k} \right).
$$

The final weight update equation becomes

$$
\begin{aligned}
\Delta w_{k+1} = {} & (1 - a_4) \cdot \Delta w_k - a_2 \cdot w_k \\
& + (a_1 \cdot e_{tot} + e'_{tot}) \cdot \left. \frac{\partial f_{out}}{\partial w} \right|_k \\
& + e_{tot} \cdot \left(\left. \frac{\partial^2 f_{out}}{\partial w \partial w^T} \right|_k \cdot w'_k + \left. \frac{\partial^2 f_{out}}{\partial w \partial f^T_{in}} \right|_k \cdot f'_{in,k} \right)
\end{aligned}
\qquad 3.19
$$

for $\Delta t = 1$, and

$$
w_{k+1} - w_k + \Delta w_{k+1} .
$$

Further reductions will be made on Eq. 3.19 for programming convenience in section 3.5. However, Eq. 3.19 represents the most descriptive form of the generalized second order approximation to Newton's Method.

3.4. Generalized Second Order Algorithm

As eluded to earlier, Parker's second order approximation is a generalized version of the steepest descent and momentum methods. From Eq. 3.19, it is readily seen that this equation contains the steepest descent search algorithm and the momentum method. The proper selection of learning parameters, a_1, ... a_5, will yield the desired algorithm.

3.4.1. Steepest Descent Algorithm

If the steepest descent search algorithm is desired, let a_1 equal a small positive number between (0, 1) and let a_4 equal 1. Set the other learning parameters equal to 0. With the above learning parameters, the single cell weight update rule is reduced to the following:

$$\Delta w_{k+1} = a_1 \cdot \left.\frac{\partial f_{out}}{\partial w}\right|_k \cdot e_{tot}$$

$$= 2 \cdot a_1 \cdot f_{out,k} \cdot (\ 1 - f_{out,k}\ ) \cdot (\ d - f_{out,k}\ ) \cdot f_{in,k}.$$

The above equation is equivalent to Lippmann's expression for the change in weights [4:17], by using the following substitutions:

$$\eta = 2 \cdot a_1,$$

$$\partial_K = f_{out,K} \cdot (1 - f_{out,K}) \cdot (d - f_{out,K})$$

such that

$$\Delta w_{K+1} = \cap \cdot \partial_K \cdot f_{in,K} \cdot$$

Hence, the algorithm of Eq. 3.19 is reduced to a gradient of steepest descent search algorithm. In addition, one of the terms of the algorithm has been explained.

3.4.2. Momentum Algorithm

In a similar fashion, the momentum algorithm is obtained. Set a_1 and a_4 to small positive numbers between (0, 1) and the other learning parameters equal to 0. Again Eq. 3.19 is reduced and the update rule becomes:

$$\Delta w_{k+1} = -a_1 \cdot \frac{\partial f_{out}}{\partial w}\bigg|_k \cdot e_{tot} + (1 - a_4) \cdot \Delta w_k .$$

Again, the above equation may be compared to the algorithm Lippmann introduces [4:17]. Let

$$\sigma = 1 - a_4$$

and again using the substitutions of section 3. 4. 1,

$$\Delta w_{K+1} = \cap \cdot \partial_K \cdot f_{in,K} + \sigma \cdot \Delta w_K \cdot$$

Again, Eq. 3.19 is reduced to obtain the momentum method and a second term of the algorithm has been identified. Appendix D offers further insight to the momentum term and possible convergence applications.

3.4.3. Additive noise

The leakage terms introduced by Parker, not only produce a momentum term to enhance convergence, but in addition, induce noise into the algorithm. The terms associated with the learning parameters, a_2 and a_3, produce the effect of noise. Recall that a_3 was encorporated into the estimated time derivative of w_K. With this in mind, a_2 and a_3 will be set to zero in all applications of this study.

3.4.4. Second Order Contributions

All of the terms of Eq. 3.19 have been described with the exception of those terms associated with a_5. The Convergence term a_5 controlls the amount of change in Δw_K and Δf_{in}. Therefore, a_5 also effects the time derivative of the total error, as seen at the end of section 3.3. Therefore, contributions from the second order derivatives and time derivatives are being implemented when a_5 is activated. If further insight is required in understanding the various components of Eq. 3.19, expansion of Eq. 3.7 may help.

3.5. Final Implementation Stage

Although Eq. 3.19 is the equation for the final weight update rule, a further reduction is necessary for a computer programmed implementation. Several temporary network variables will be defined to help simplify the programming overhead. Each node in every layer is responsible for performing two passes; a forward pass and backward pass. This fact will be used in associating the temporary variables with the model and further reducing the programming overhead.

In Eq. 3.19 there are several partial derivatives which must be computed and reduced to a form acceptable for programming. In other words, what is the partial of f_{out} with respect to $\mathbf{w}$? What is the second partial of f_{out} with respect to $\mathbf{w}$ and $\mathbf{w}^T$ and also $\mathbf{w}$ and $f_{in}{}^T$? These partials must be computed in order to simplify the programming model. Appendix C is devoted solely to the computations of the various partials of the sigmoid function and the results will be used here.

$$\left.\frac{\partial f_{out}}{\partial \mathbf{w}}\right|_k : f_{out,k}\cdot(1 - f_{out,k})\cdot \mathbf{f}_{in,k}$$

$$\left.\frac{\partial^2 f_{out}}{\partial \mathbf{w}\partial \mathbf{f}^T_{in}}\right|_k : f_{out,k}\cdot(1 - f_{out,k})\cdot \mathbf{I}$$
$$+ f_{out,k}\cdot(1 - f_{out,k})\cdot(1 - 2\cdot f_{out,k})\cdot \mathbf{f}_{in,k}\cdot \mathbf{w}^T$$

$$\left.\frac{\partial^2 f_{out}}{\partial \mathbf{w}\partial \mathbf{w}^T}\right|_k : f_{out,k}\cdot(1 - f_{out,k})\cdot(1 - 2\cdot f_{out,k})\cdot \mathbf{f}_{in,k}\cdot \mathbf{f}^T_{in,k}$$

Applying these equations to the algorithm directly would be quite cumbersome, hence the temporary variables. Before substituting the above equations into Eq. 3.19, the following temporary variable is defined. Let

$$u : f_{out,k}\cdot(1 - f_{out,k})$$

where u is a common artifact generated when computing the partial derivatives of f_{out}. Next, substitute the definitions of each partial, along with u, into Eq. 3.19, such that

$$\begin{aligned}\Delta\mathbf{w}_{K+1} = {} & (1 - a_4)\cdot\Delta\mathbf{w}_K - a_2\cdot\mathbf{w}_K \\ & + (a_1\cdot e_{tot} + e'_{tot})\cdot u\cdot\mathbf{f}_{in,K} \\ & + e_{tot}\cdot\Big(u\cdot(1 - 2\cdot f_{out,K})\cdot\mathbf{f}_{in,K}\cdot\mathbf{f}^T_{in,K}\cdot\mathbf{w}'_K \\ & \qquad + (u\cdot I + u\cdot(1 - 2\cdot f_{out,K})\cdot\mathbf{f}_{in}\cdot\mathbf{w}^T)\cdot\mathbf{f}'_{in,K}\Big)\end{aligned}$$

$$\begin{aligned}\Delta\mathbf{w}_{K+1} = {} & (1 - a_4)\cdot\Delta\mathbf{w}_K - a_2\cdot\mathbf{w}_K + (a_1\cdot e_{tot} + e'_{tot})\cdot u\cdot\mathbf{f}_{in,K} \\ & + e_{tot}\cdot\Big(u\cdot(1 - 2\cdot f_{out,K})\cdot\mathbf{f}_{in,K}\cdot\Big(\mathbf{f}^T_{in,K}\cdot\mathbf{w}'_K + \mathbf{w}^T\cdot\mathbf{f}'_{in,K}\Big) \\ & \qquad + u\cdot\mathbf{f}'_{in,K}\Big)\end{aligned}$$

In the above equation, let

$$v = \mathbf{f}^T_{in,K}\cdot\mathbf{w}'_K + \mathbf{w}^T\cdot\mathbf{f}'_{in,K}$$

where v represents the time derivative of the product of the input and it's corresponding weight, such that

$$\begin{aligned}\Delta\mathbf{w}_{K+1} = {} & (1 - a_4)\cdot\Delta\mathbf{w}_K - a_2\cdot\mathbf{w}_K + (a_1\cdot e_{tot} + e'_{tot})\cdot u\cdot\mathbf{f}_{in,K} \\ & + e_{tot}\cdot\Big(u\cdot(1 - 2\cdot f_{out,K})\cdot\mathbf{f}_{in,K}\cdot v + u\cdot\mathbf{f}'_{in,K}\Big).\end{aligned}$$

Next, define

$$q = u \cdot e_{tot}$$

where q is the ∂ term defined in sections 2.3.5 and 3.4.1 and the weight update equation becomes:

$$\Delta w_{K+1} = (1 - a_4) \cdot \Delta w_K - a_2 \cdot w_K + a_1 \cdot q \cdot f_{in,K} + u \cdot e'_{tot} \cdot f_{in,K}$$
$$+ u \cdot e_{tot} \cdot (1 - 2 \cdot f_{out,K}) \cdot f_{in,K} \cdot v + q \cdot f'_{in,K} \cdot$$

Finally, let

$$r = u \cdot (e_{tot} + e'_{tot} \cdot (1 - 2 \cdot f_{out,K}) \cdot v)$$

and the final weight update equation becomes:

$$\Delta w_{K+1} = (1 - a_4) \cdot \Delta w_K - a_2 \cdot w_K + (a_1 \cdot q + r) \cdot f_{in,K} + q \cdot f'_{in,K} \cdot \qquad 3.20$$

Equation 3.20 represents the final form. Obviously, each temporary variable will be computed first before Eq. 3.20 is computed.

3.5.1 Forward Pass

Parker describes the flow of signals in two directions [8]. The input enters the bottom of the network and flows forward with each node in each layer computing what Parker calls function signals. Function signals represent the cell outputs and their respective time derivatives. The cell outputs become the cell inputs in the layer directly above the cell in question. The output time derivatives become the input time derivatives in a similar manner. The output layer produces a function signal which is immediately compared to some desired response. The partial derivative of the squared difference between the desired and actual outputs is termed the error signal. The error signal is treated in a similar manner as the function signals, but they are back propagated. In addition, the time derivative of the error signals is computed from the time derivative of the output and propagates with the error signal.

On the forward pass each cell in each layer is responsible for computing and maintaining it's own copy of the following:

$$f_{out,k} = f_{out,k}(f_{in,k}, w_k) = \frac{1}{1 + \exp(-f^T_{in,k} \cdot w_k + \Theta)},$$

$$u = f_{out,k} \cdot (1 - f_{out,k}),$$

$$w'_k = a_3 \cdot w_k + a_5 \cdot \Delta w_k,$$

$$v = f^T_{in,k} \cdot w'_k + w^T \cdot f'_{in,k}$$

The threshold (Θ) is the cell offset and is updated much in the same way the weights are updated.

3.5.2. Backward Pass

As mentioned earlier, the cell must also be capable of handling the backward propagation functions, see Fig. 3.1.

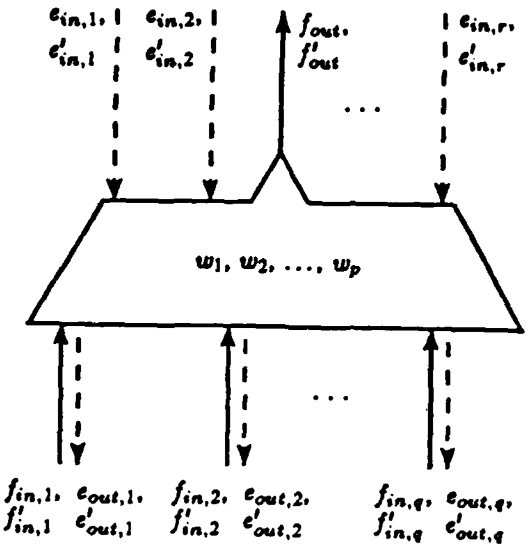


Figure 3.1 Signal Flow Through a Single Cell [8]

Therefore on the backward pass, the cell must compute the error signals as well as their time derivatives. Figure 3.2 presents a simple two layer network for enhancing the discussion below.

Since the algorithm used for this thesis effort is a supervised backprop, there must be some desired response from which to compare the actual response. The output of each node in the output layer will be used to compute an error signal used for updating the weights. Recall that the function of backprop is to back propagate the partial derivatives of the quantity to be minimized. First, the total error and it's approximate time derivative received by each cell are computed, where

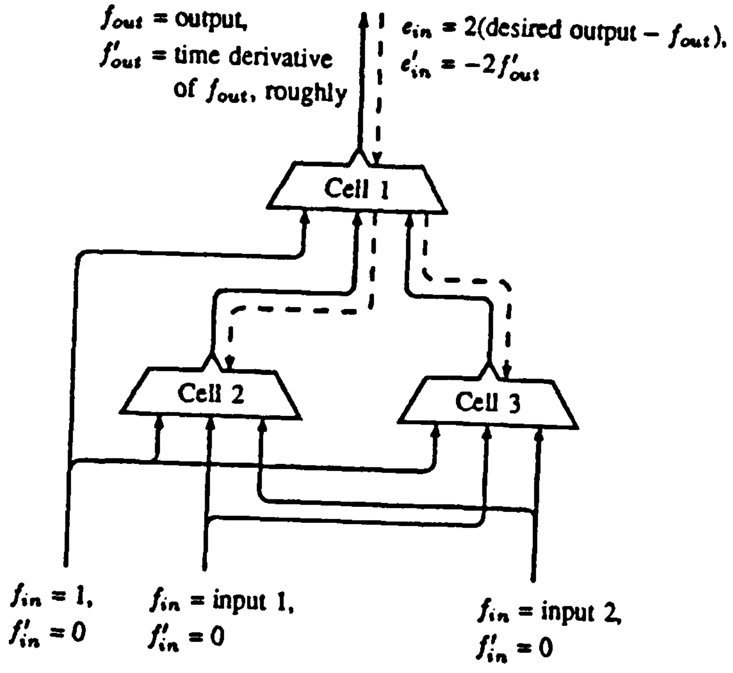

f_{out} = output,
f'_{out} = time derivative of f_{out}, roughly
e_{in} = 2(desired output − f_{out}),
$e'_{in} = -2f'_{out}$
Cell 1
Cell 2
Cell 3
f_{in} = 1, f'_{in} = 0
f_{in} = input 1, f'_{in} = 0
f_{in} = input 2, f'_{in} = 0

Figure 3.2 Two Layer Network Display

$$e_{tot} = \sum e_{in,\,i},$$

$$e'_{tot} = \sum_{i=1}^{r} e'_{in,\,i}.$$

The second step is to compute the output error signal and it's time derivative for propagation to the next lower layer. The output error is the product of the partial and the corresponding weight and can be found by,

$$e_{out} = e_{tot} \cdot \left.\frac{\partial f_{out}}{\partial f_{in}}\right|_{k} = e_{tot} \cdot f_{out,k} \cdot (1 - f_{out,k}) \cdot w_{k} = q \cdot w_{k},$$

where the results of appendix C and the temporary variables have been applied. The partial time derivative of e_{out} is found in much the same way and applying the chain rule, such that

$$e'_{out} = e'_{tot} \cdot \frac{\partial f_{out}}{\partial f_{in}} + e_{tot} \cdot \left(\left.\frac{\partial^2 f_{out}}{\partial f_{in} \partial w^T}\right|_{k} \cdot w'_{k} + \left.\frac{\partial^2 f_{out}}{\partial f_{in} \partial f^T_{in}}\right|_{k} \cdot f'_{in,k} \right).$$

$$= r \cdot w_K + q \cdot w'_K.$$

Again, see appendix C and the temporary variable definitions for clarity.

Once e_{out} and it's partial time derivative have been defined, they can then be related to e_{in} of the cells of the next lower layer and the process is repeated. The cell's weights may be updated layer by layer during backprop or the results may be stored and updated after the backward pass is complete.

NOTE: For ease of programming and efficiency of the code, each layer in the multilayer perceptron will be treated as a record, since each layer has common attributes. By attributes, it is implied that each layer will have a vector of outputs, and a matrix of weights. Remember, that the inputs to a cell can originate from the environment or from the other cells in the layer directly beneath it as outputs from that layer. Therefore, the inputs will be considered vectors as well. The error signals associated with each cell will also be defined as a vector describing the error signals of a given layer.

3.6. Initial Network Conditions

A great deal of discussion has gone into the implementation stages of the second order back propagation algorithm. However, the questions that arise are: what is the state of the network when training is initialized? What initial values are assigned to the weights, thresholds and input partial time derivatives? This section addresses each of these questions.

In answering the question of the initial values of the weights and thresholds, it's desired to have the activation level of each cell roughly equal to 0. Thus,

$$\mathbf{f}_{in}^{T} \cdot \mathbf{w} + \Theta \approx 0$$

implies that each cell within the network fires at approximately 0.5. The reason this is so important, is that if the output cells fire close to 1 or 0 early in training and the desired output is 0 or 1 respectively, the network may never recover and successfully train. Consider the following argument given a first order minimization technique. The weight update rule for the gradient of steepest descent has the following form from section 3.4.1:

$$\Delta w_{k+1} = 2 \cdot a_1 \cdot f_{out,k} \cdot (1 - f_{out,k}) \cdot (d - f_{out,k}) \cdot f_{in,k} \cdot$$

Consider the output from a given cell in the output layer is ~1 and the desired is equal to 0. The desired minus the output is ~-1, but the other difference term is ~0. Therefore, under these conditions little or no change occurs in the weights. However, if each cell in the network is initially firing near 0.5, the network is provided the opportunity to learn.

More information can be provided by examininig the second partial of s with respect to **w** and $\mathbf{w}^T$. The underlying idea is to examine the sign of definiteness of this matrix over the entire ensemble of training vectors. If it is possible to show that the average second partial matrix is positive definite over the entire input ensemble, then this would imply a surface with upward concavity. This further implies a global minimum over the entire ensemble of input vectors considered. Appendix D considers this for a single cell and establishes a criterion for initializing training in a neighborhood of the global minimum for an arbitrary training set. The result is provided here:

$$-\ln(2) \le \sum_{i=1}^{n} w_i \cdot f_{in,i} \le \ln(2). \qquad 3.21$$

It is implied in the above inequality that one of the inputs is equal to 1, corresponding to the so called threshold.

Information about the input may reduce the above equation to strictly a function of the weights. For instance, the FLIR feature vectors were normalized according to Eq. 2.1. After normalization, the features ranged from (~-1.5, ~1.5). Selecting a worst case scenario where all the features of a given vector equal 1.5 (or -1.5), the above criterion reduces to:

$$-0.462 \le \sum_{i=1}^{n} w_i \le 0.462.$$

The above inequality of Eq. 3.21 provides a measure as to the initial weight settings for a given cell. For instance, the weights could be set randomly and uniformly between $(-\tau, \tau)$, where τ is a small floating point number, if some information is known about the input. A test could then be performed to insure that the above inequality is met. Randomly, setting the weights uniformly between (-0.45, 0.45) for the input data in this study satisfied the above criterion, provided by Eq. 3.21.

The final question to be answered considers the time derivatives of the input from the environment. No information was provided concerning the time derivatives of the input from the environment. Since the algorithm is a discrete version of the second order linear differential equation, it will be assumed that each input will be constant over the period of time (Δt) in question. Therefore, it is assumed that the network input time derivatives are equal to 0. It should not be assumed, however, that the input to the hidden layer nodes are zero. Recall, that the output of each layer becomes the input of the cells in the above layer. It is understood that the output is changing with time, as per section 3.4.

3.7. Summary

The first several sections of this chapter is devoted to the impementation of Parker's linear differential equation in Eq. 2.8. A great deal of text was devoted towards the implementing stages to achieve a better understanding of the concepts hidden within the mathematics. Time derivatives of the signals were derived in order to inform the network about how the performance surface is changing with time. With the information in this chapter, along with appendices A, B, C, and D, most (if not all) of the concepts have surfaced. Once all of the terms of the final implementing equation had been derived, it was necessary to introduce the temporary variables to ease the programming effort. Finally, initial network conditions were considered.

The following chapter provides the results of the validation stage of the algorithm discussed in section 3.3. Chapter four begins by using the algorithm derived in this section to solve the exclusive or problem. The latter sections test the algorithm on the doppler imagery which has already been classified by Ruck, to further validate the algorithm.

4. Validation of Second Order Algorithm

4.1. Introduction

In the last chapter, an extensive analysis of the second order (SO) algorithm implementation was realized. In this chapter, it is desired to focus on the validation stage of the classifier applying this new algorithm. The validation stage will consist of two parts. The first involves application of the SO back propagation algorithm in a network used to solve the exclusive OR (XOR) problem. The second stage initiates the quest of pattern classification beginning with set of feature vectors generated from doppler imagery. Ruck used these same features with moderate success [12]. He was able to attain near perfect classification with the training data and roughly 75% classification of the test data [12]. These facts add validity to the input feature vectors. The generalized second order algorithm developed in this study, will allow the comparison between first and second order techniques.

The next section begins with a description of the XOR problem set up. The input and learning parameters used are provided in this section, as well as the convergence results. Section 4.3 begins the pattern classification effort for this study. Within this section, the input feature vectors are described and formatted. In addition, the network architecture and learning parameters are described for the gradient of steepest descent, momentum, and second order methods. The results of the pattern classification of the doppler imagery feature vectors follow.

4.2. The Exclusive OR Problem

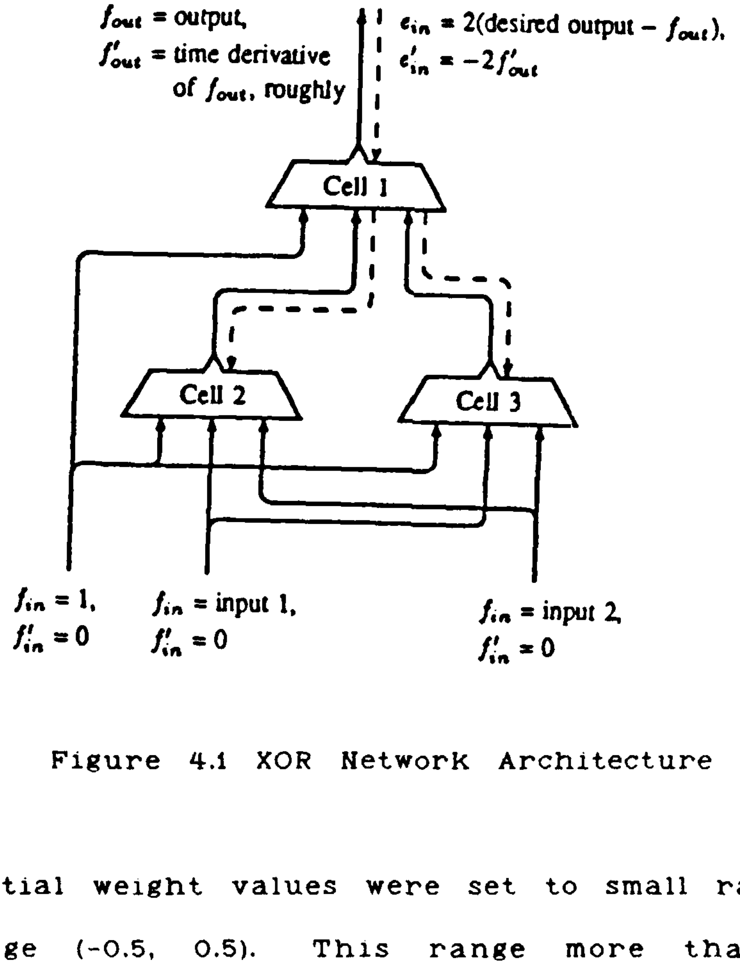
The results of this section are basically a reproduction of the results generated by Parker [8]. Since the algorithm used for this study generalizes to the first order methods of steepest decent and momentum, the problem will be attempted using both second and first order techniques. The results will be formulated in a table based on the number of iterations until convergence.

4.2.1 Input Data and Network Parameters

The idea is to train the network on a fixed set of inputs which are listed in Table 4.1, along with the desired responses. The inputs will be shown to the network as in Fig. 4.1, iteratively until the desired outputs are obtained. The output will be measured indirectly by monitoring the error. The error is defined as the difference between the desired output and the actual output. Therefore, the criterion used in validating the model will be the error. By minimizing the error, the squared error will surely follow.

Table 4.1 Input Pattern Vectors and Desired Response for XOR

Vector	$f_{in,1}$	$f_{in,2}$	Desired Response
1	0. 1	0. 1	0. 1
2	0. 9	0. 1	0. 9
3	0. 1	0. 9	0. 9
4	0. 9	0. 9	0. 1



f_{out} = output,
f'_{out} = time derivative of f_{out}, roughly
e_{in} = 2(desired output − f_{out}),
$e'_{in} = -2f'_{out}$
Cell 1
Cell 2
Cell 3
f_{in} = 1, f'_{in} = 0
f_{in} = input 1, f'_{in} = 0
f_{in} = input 2, f'_{in} = 0

Figure 4.1 XOR Network Architecture

The initial weight values were set to small random numbers in the range (-0.5, 0.5). This range more than meets the criterion provided by Eq. 3.21. The contents of Table 4.2 list the values of the learning parameters used in solving this problem.

An extensive search of the optimum learning parameters was not performed. It was only desired to prove that the network and it's SO back propagation algorithm could solve the XOR problem. By solving the problem, it is implied that the network found the optimum path for the weights to follow. When the error is reduced to some predetermined criterion, the network concludes it's training, and the optimum weight values are obtained. The error criterion or the difference between the desired and actual output was set to 0.1. The criterion must be met or surpassed on four successive iterations, allowing the network the opportunity to classify all four inputs and meet the criterion.

Table 4.2 Learning Parameter Values

Method	a_1	a_2	a_3	a_4	a_5
Gradient	0. 1	0. 0	0. 0	1. 0	0. 0
Momentum	0. 1	0. 0	0. 0	0. 1	0. 0
Second Order	0. 1	0. 0	0. 0	0. 1	0. 05

4.2.2 Convergence Results

The results tabulated in Table 4.3 show that the network weights found the optimum path for convergence. This implied that the network can perform the XOR logic function. The average number of iterations were generated from 20 test runs.

Table 4.3 Comparison Between First and Second Order Techniques

Method	Gradient	Momentum	Second Order
Average Number of Iterations	>20, 000	5474	5054

The results show that on average, the SO method slightly out performed the momentum method. In addition, both methods greatly exceeded the performance of the gradient of steepest descent method. The gradient of steepest descent method was extremely slow in learning and terminated after 20,000 iterations, with the error slowly decreasing.

The results appear to be very promising for extending the application of the SO approximation to more difficult problems. The following section addresses such a problem. The fact that the SO approximation method exceeded the performance of the first order methods, in no way suggests that it will exceed performances on more difficult problems. In particular, when considering the problem of pattern classification where the inputs may be great in number. Other considerations along this same line, are the number of layers required, the number of nodes and ultimately the number of weights required to solve the problem of machine recognition of images.

The ADA programming code used in implementing the XOR algorithm is found in appendix F.

4.3. Classification of Doppler Imagery

In the last section it was shown that the SO approximation method could in fact be used in solving the XOR problem. The results represent a promising indication that the applications may be extended using this method. Therefore, in this section, features extracted from doppler imagery will be used as the input to a multilayer perceptron. The multilayer perceptron will apply the SO back propagation method, as well as the first order methods for comparison.

The following subsection describes the features in a little more detail. Next, the specifics of the network architecture are discussed, followed by some comments on the values used for the learning parameters. The final subsection discusses the results and makes a comparison between the first and second order back propagation techniques.

4.3.1 Input Feature Data

The features extracted from the doppler imagery consisted of normalized moment invariants. To the network, the features were actually a set of vector components of normalized moments. Each vector, or example of a target, consisted of 22 features, and in general, the final version of a machines representation of an object in an image. The targets to be classified included tanks at four different aspect angles, jeeps, 2.5 ton trucks and petroleum, oil, and lubricant (POL) tankers. The data base of target feature vectors heavily favored the tanks. Table 4.4 breaks down the number targets considered for each class.

Table 4.4 Target Data Base for Classification

Class	# Training Samples	# Test Samples
Tank	43	17
POL	4	2
Jeep	6	3
Truck	4	2

Roughly two thirds of the available feature vectors were used in training the network, while the remaining vectors were used for testing the network once trained.

4.3.2. Network Architecture and Learning Parameters

The multilayer perceptron will consist of three layers, which will accept 22 inputs and output 4 classes. Table 4.5 describes the network architecture in some detail. Table 4.6 provides the learning parameters used for classifying the doppler imagery feature vectors. Several different combinations of learning parameters were used for training. This search was not exhaustive, since there is an enormous number of these combinations. However, the parameters listed in Table 4.6 provided the best combination, of those tried, as far as classification accuracy and error performance were concerned.

Table 4.5 Network Architecture Data

Number of Features	22
Layer One Nodes	20
Layer Two Nodes	6
Number of Classes	4

Table 4.6 Network Training Data

Parameter	Gradient Method	Momentum Method	SO Method
a_1	0.3	0.3	0.3
a_2	0.0	0.0	0.0
a_3	0.0	0.0	0.0
a_4	1.0	0.1	0.1
a_5	0.0	0.0	0.1
Number of Iterations	60,000	60,000	60,000
Data Output Interval	2,000	2,000	2,000

The text to follow provides the results generated from target classification of the doppler imagery. Average classification accuracy and the average total output error is provided, along with the network performance on individual classes.

4.3.3. Classification Results

The graphs depicted below are the results from the classification of the feature vectors generated from doppler imagery. The results are displayed in terms of average classification accuracy versus the number of iterations for both the first and second order methods. The log of the average total output error versus the log of the number of iterations was measured, as well. In addition, a typical instance of classification accuracy for each class is listed below. An instance implies that the data was not averaged. The graphs below are presented in order of test results rather than by method, for ease of comparing each method.

4.3.3.1 Average Classification Accuracy

The average classification accuracy was taken from 10 complete passes through the network, since it was desired to obtain an average network performance. Given the randomness of the initial state of the network, the network does not perform in exactly the same way with each training attempt. Each pass through the network will re-initialize the network parameters and begin training all over again. The specific characteristics desired for comparison were the convergence rate and stability of each method.

The criterion used in determining a correct response, and thus the accuracy of the classifier, was based on the actual output values of the nodes in the output layer. The desired node output for a correct classification is 1, while all the other node outputs are 0. Therefore, the criterion defines the output response of the desired node to fire at 0.8 or above, while the other nodes fire at 0.2 or less.

Figures 4.2, 4.3 and 4.4 display the average training accuracy of the gradient of steepest descent, momentum and second order methods respectively. Figures 4.5, 4.6 and 4.7 display the average test accuracy.

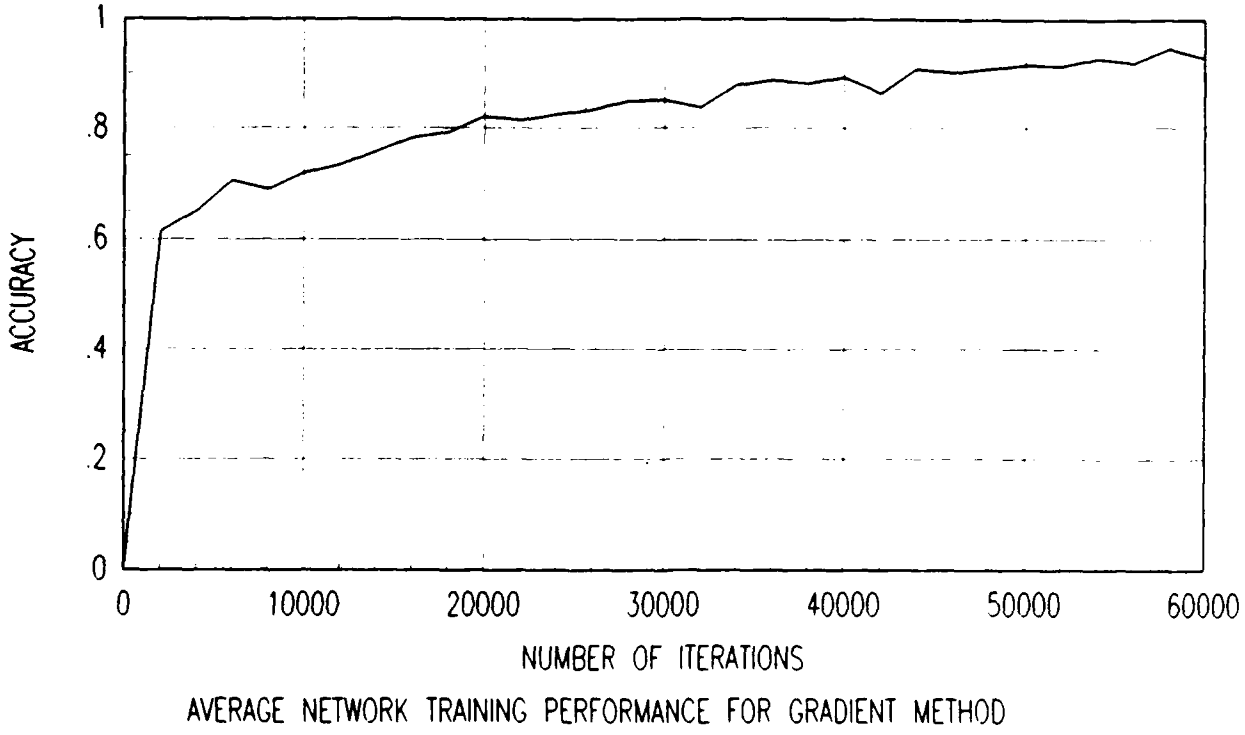


Figure 4.2 The network achieved roughly 92% accuracy on training data.

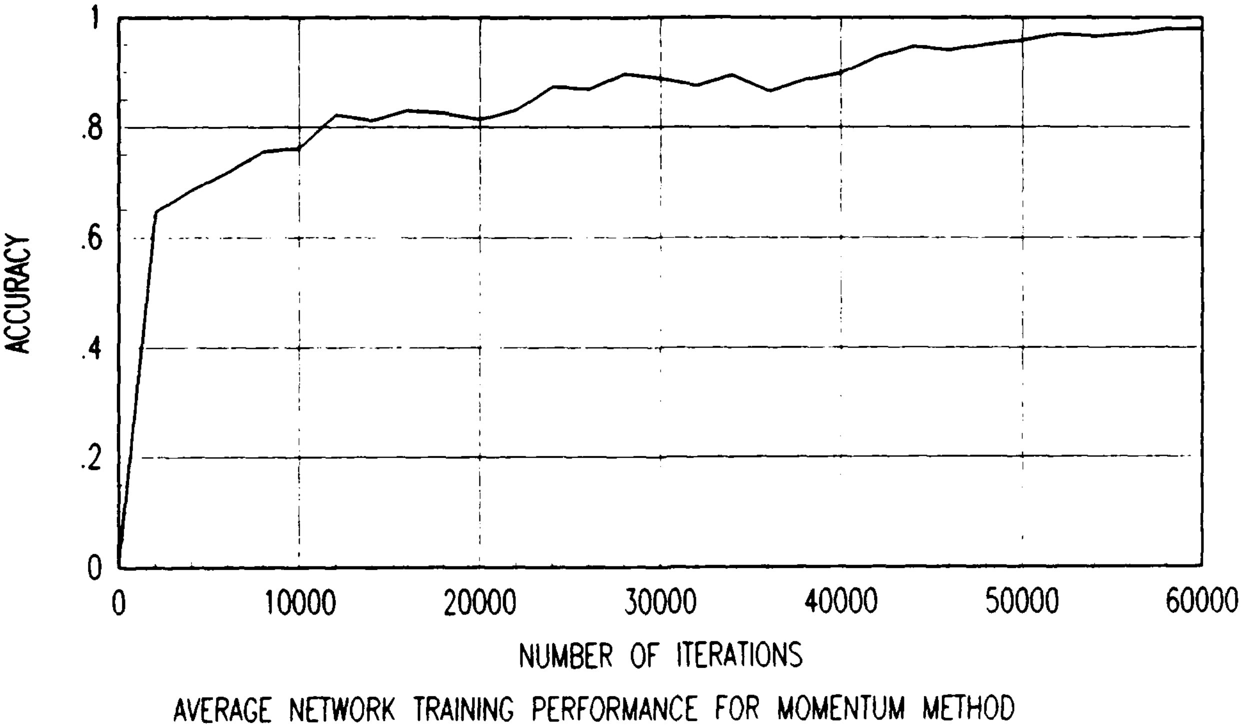


Figure 4.3 The network achieved 98% accuracy on training data.

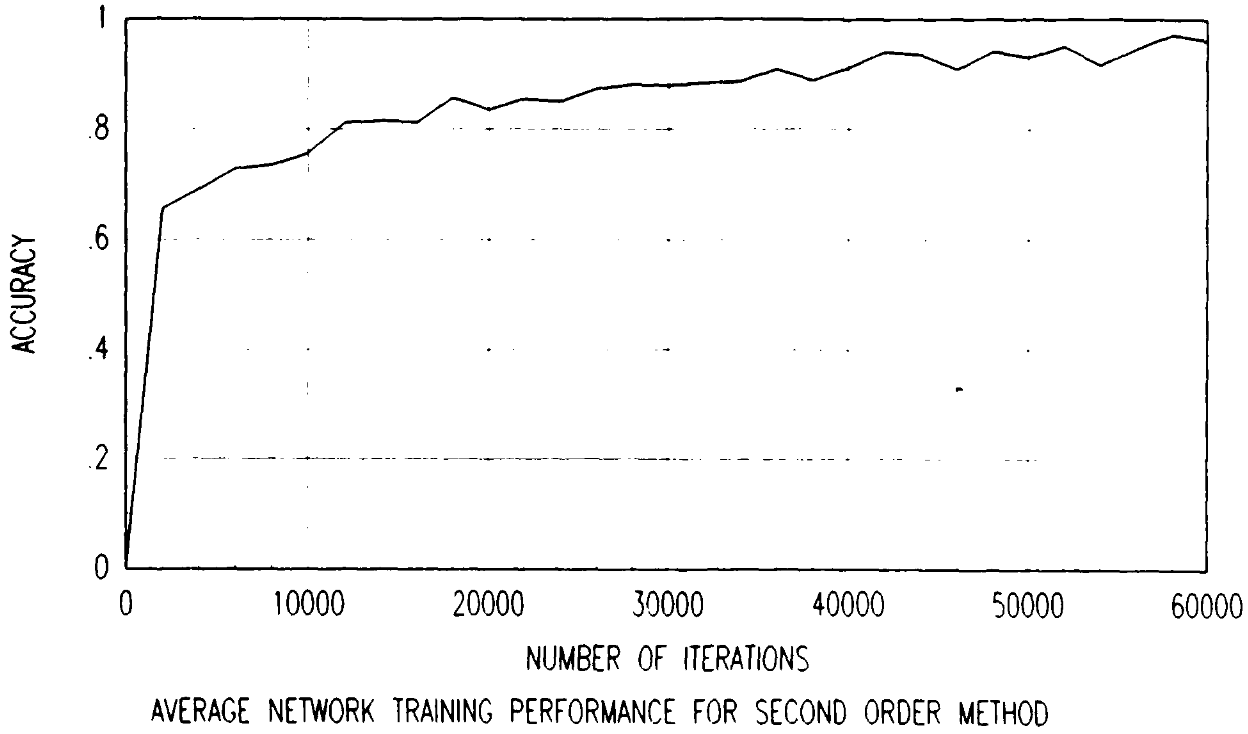


Figure 4.4 The network achieved 98% accuracy on training data.

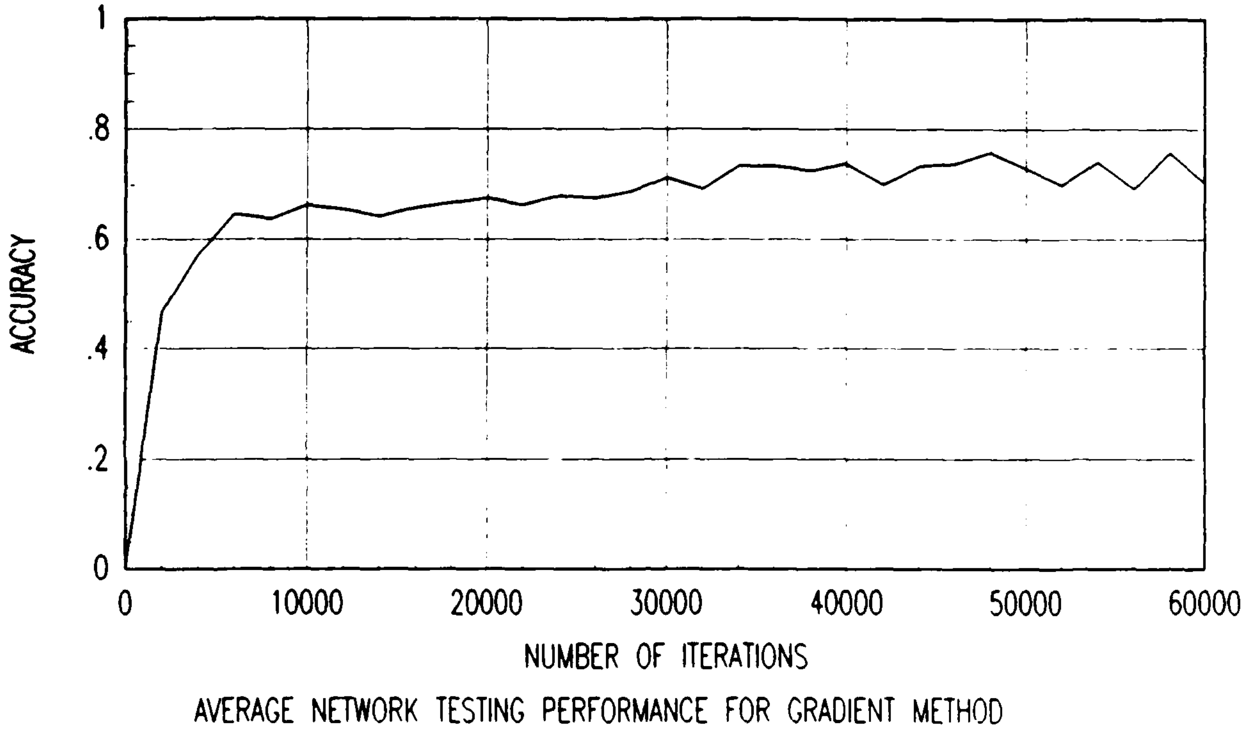


Figure 4.5 The network achieved 75% accuracy on the test data.

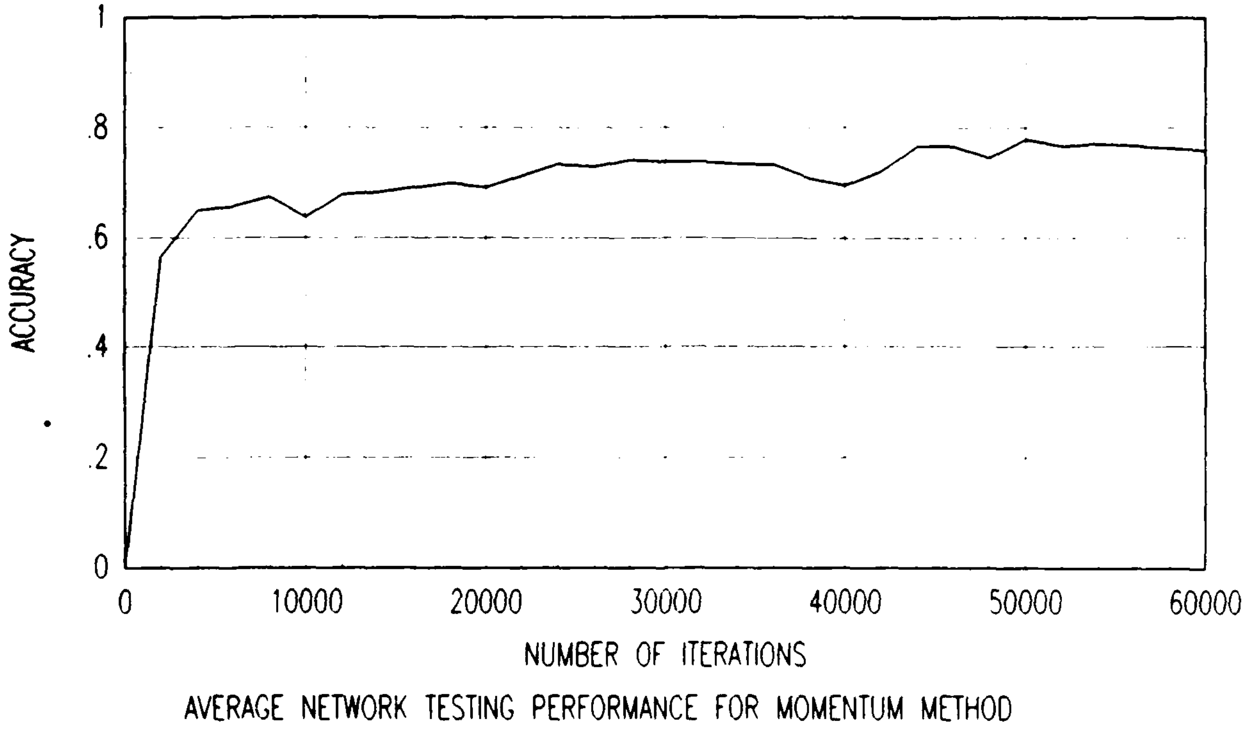


Figure 4.6 The network achieved 78% accuracy of test data.

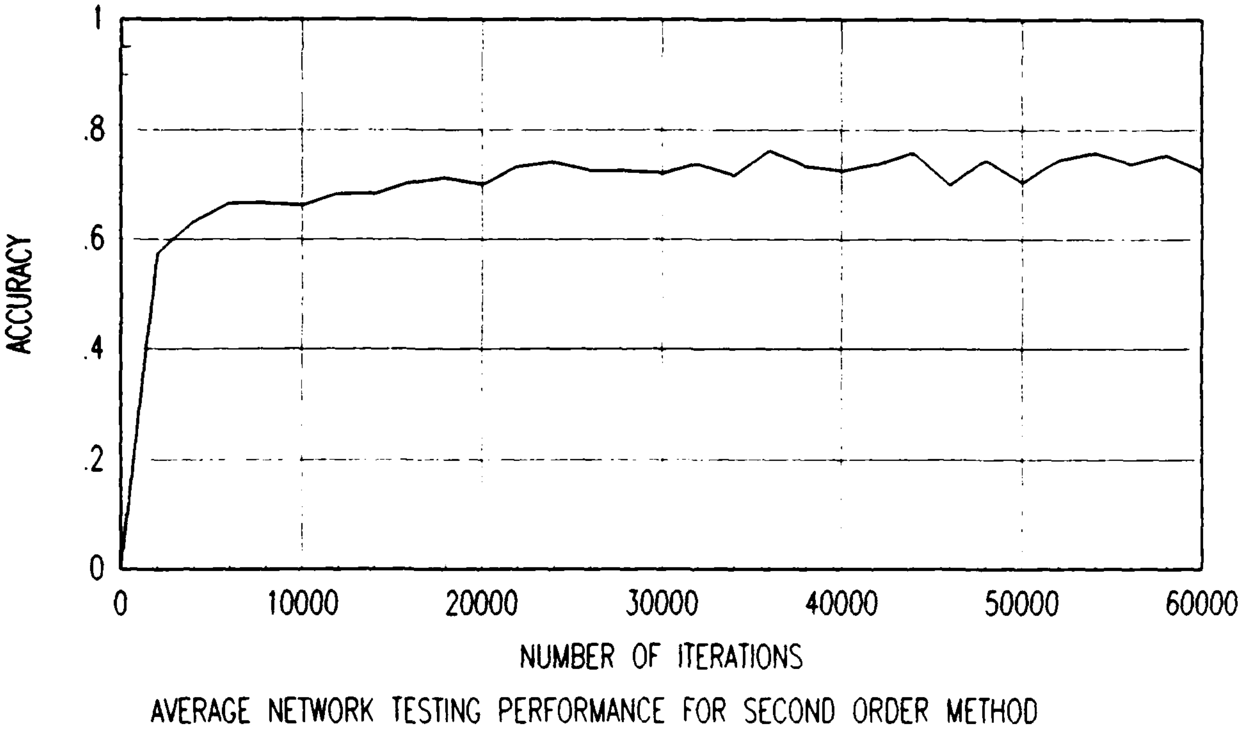


Figure 4.7 The network achieved 78% accuracy of test data.

The momentum and second order methods slightly exceeded the performance of the gradient of steepest descent method on the test and training data. On the average, there was little difference between the momentum and second order methods. However, close examination of the average classification accuracy reveals that the second order method initially converges slightly quicker than the momentum method. Over the last 10,000 iterations the momentum method seemed to settle down and provide a consistent accuracy. On the other hand, the second order method continued to climb, but in a slightly erratic manner.

4.3.3.2 Average Total Output Error

This section presents the results of the average total output error of the network. The error was defined to the magnitude of the difference between the desired output and the actual output. The total output was merely the error sum of all of the output nodes. The total output error was averaged over the entire set of training (or testing) input feature vectors, and ultimately over each pass through the network. The log of the average error was graphed versus the log of the number of iterations. It was desired to determine what trends, if any, the average error displayed for both the training and test data sets. Again, Figs. 4.8, 4.9 and 4.10 reflect the training results of the gradient, momentum and SO methods, respectively. Figures 4.11, 4.12 and 4.13 display the results of the test data.

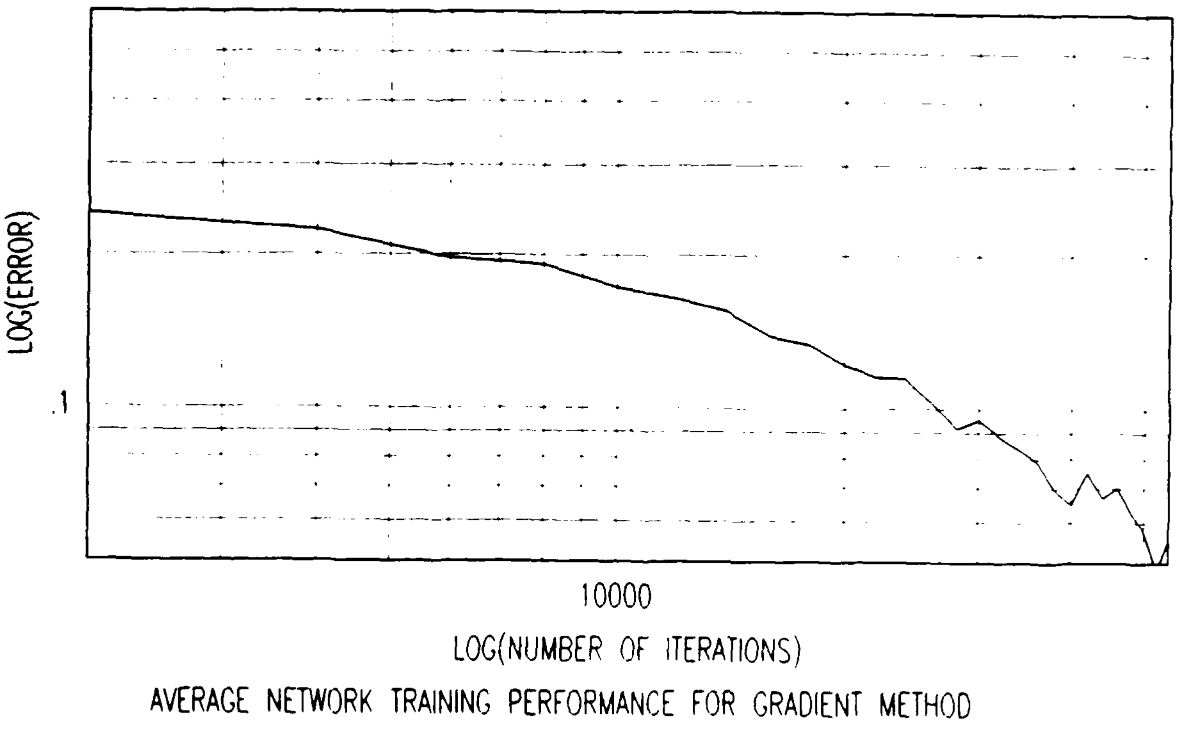


Figure 4.8 Note the smooth (almost monotonic) decreasing error.

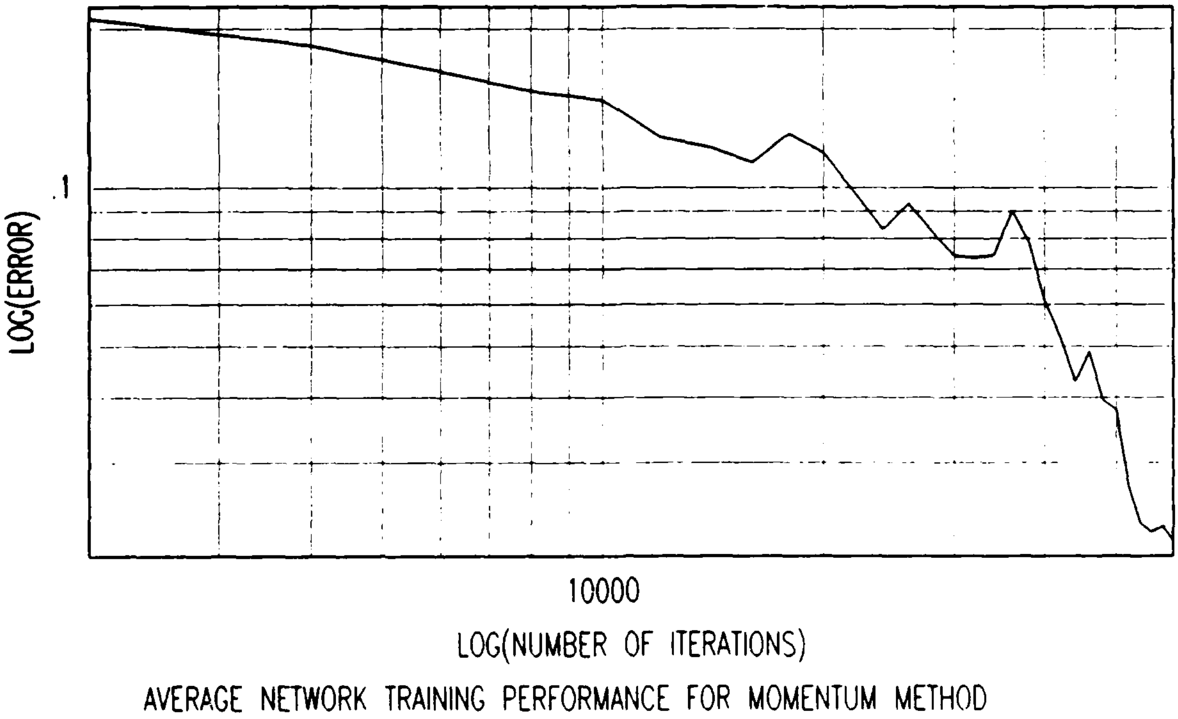


Figure 4.9 The initial error is dropping off smoothly, but becomes erratic as training continues.

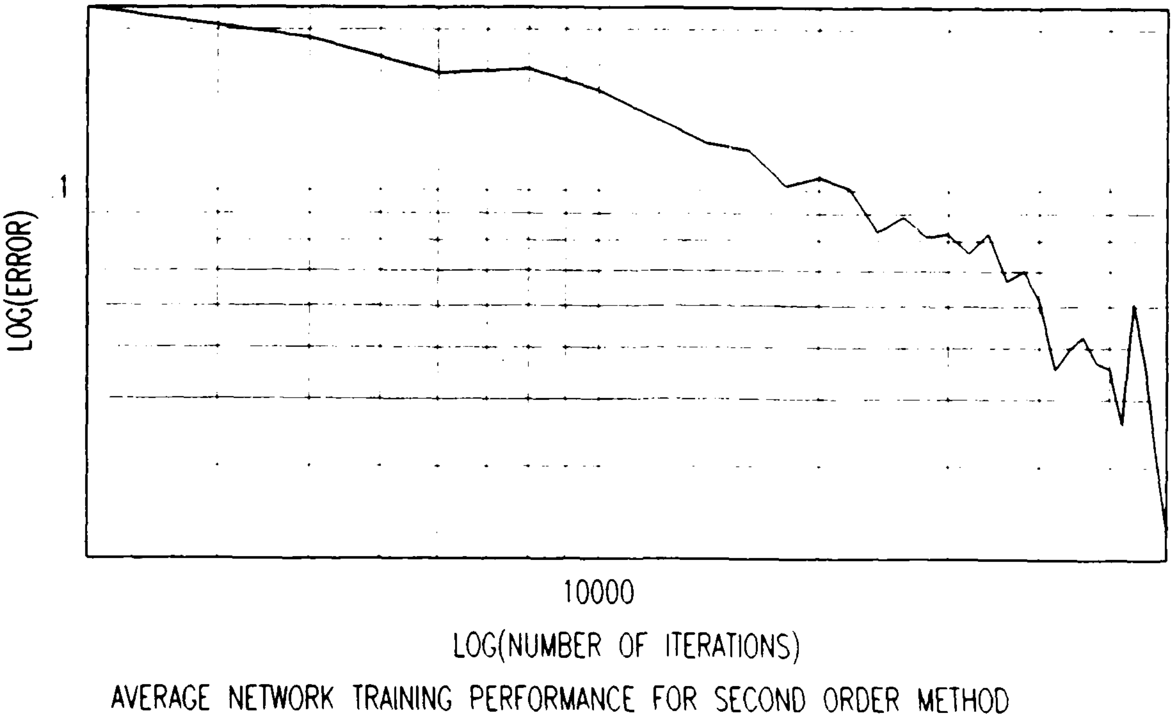
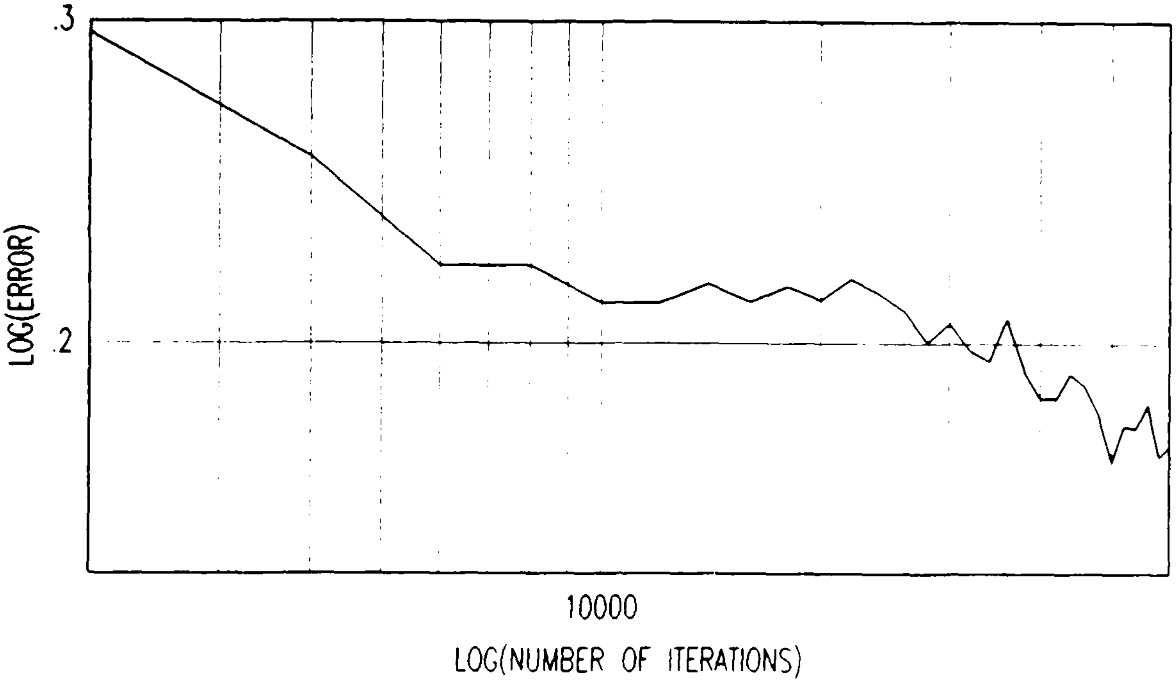
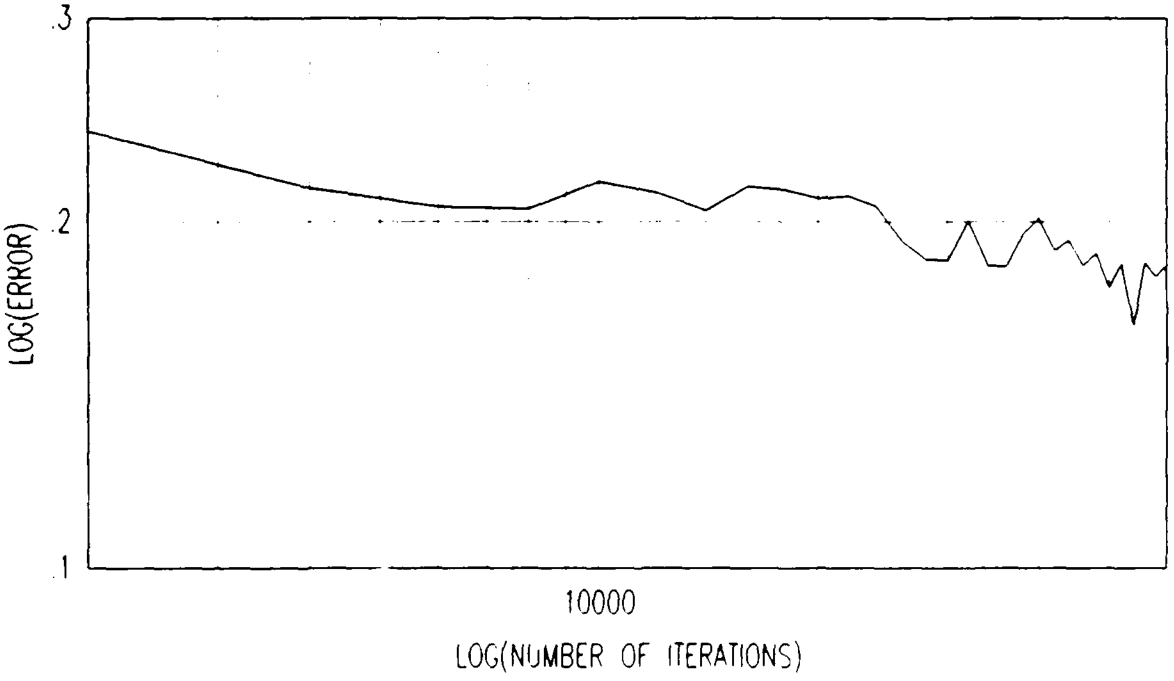


Figure 4.10 The initial error is dropping off smoothly, but becomes erratic as training continues.



AVERAGE NETWORK TESTING PERFORMANCE FOR GRADIENT METHOD

Figure 4.11 Note the initial smooth descent. In addition, notice that the relative flat region in the middle and then a slightly erratic descent over the last 30,000 iterations.



AVERAGE NETWORK TESTING PERFORMANCE FOR MOMENTUM METHOD

Figure 4.12 The initial error drops much quicker then the gradient method and becomes erratic as training continues.

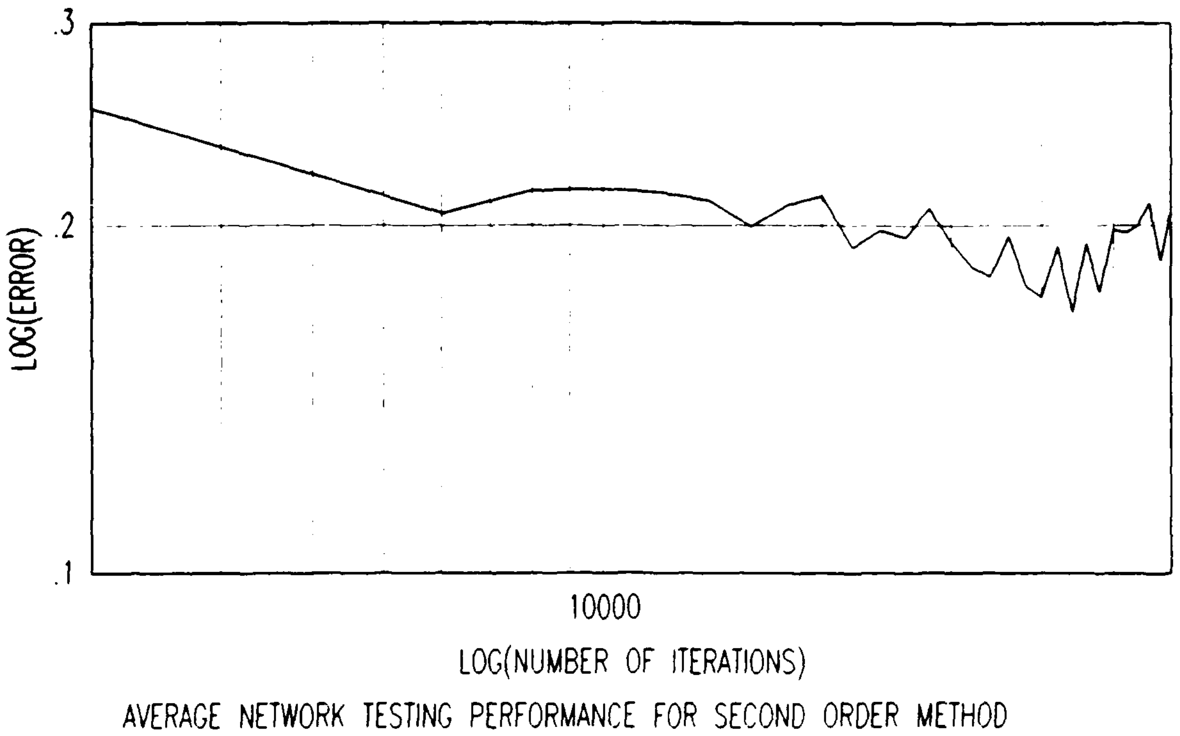


Figure 4.13 The initial error drops much quicker then the gradient method and becomes erratic as training continues.

Again, the momentum and second order methods slightly exceed the average error performance of the gradient of steepest descent method. In addition, the momentum and second order methods for the most part performed on a comparable basis. Over the first half of training, the error decreases rather smoothly, after which the error behavior of the momentum and second order methods becomes erratic. This may be explained by the following argument. If the minimum of the error surface lies in a relatively flat hyperplane, then there could be many solutions (optimum weight values) for convergence. Since the instantaneous error surface is changing with each input, the network is simply trying to find an exact solution. This may be interpreted, as the network attempting to memorize the input data. The classification results of the FLIR imagery in the following chapter further support this idea. The gradient of steepest descent method has a little more stability over the last half of training, as shown in Figs. 4.8 and 4.11. The reason for this, is that the weights are being updated by a small constant proportion of the partial derivative. This is slow gradual process and the network has not quite converged to the minimum; it's still learning.

4.3.3.3 Target Accuracy

The data in this section reflect a typical instance of the actual target accuracy provided by all three methods. Again training and test data results are displayed the tables below. The data gathered and displayed in the tables below were actually generated from the network output. The tables are presented in the form of a Network Confusion Matrix. Not only was it desired to determine the actual target accuracy, but this data may provide some insight as to the worth of the feature vectors discriminating the various targets.

Table 4.7, 4.8, and 4.9 list the results from the training data, while Table 4.10, 4.11, and 4.12 display the results form the test data.

Table 4.7 Training Data Confusion Matrix for Gradient Method. Reads row by row left to right. The network failed to classify the following number of feature vectors: 2 tanks and 1 each POL, jeep, and truck.

Class	Tank	POL	Jeep	Truck	Accuracy
Tank	41	0	0	0	95. 3%
POL	0	3	0	0	75%
Jeep	0	0	5	0	83. 3%
Truck	0	0	0	3	75%

Table 4.8 Training Data Confusion Matrix for Momentum Method. Reads row by row left to right. The network failed to classify 1 POL feature vector.

Class	Tank	POL	Jeep	Truck	Accuracy
Tank	43	0	0	0	100%
POL	0	3	0	0	75%
Jeep	0	0	6	0	100%
Truck	0	0	0	4	100%

Table 4.9 Training Data Confusion Matrix for Second Order Method. Reads row by row left to right. The network failed to classify 1 POL feature vector.

Class	Tank	POL	Jeep	Truck	Accuracy
Tank	43	0	0	0	100%
POL	0	3	0	0	75%
Jeep	0	0	6	0	100%
Truck	0	0	0	4	100%

The training results in all the tests continue to enhance the idea, that if a network is trained long enough, it will learn or at the very least memorize the input training set.

Table 4.10 Test Data Confusion Matrix for Gradient Method. Reads row by row left to right. Notice that two jeep feature vectors were classified as a tank, while the other was classified as a truck.

Class	Tank	POL	Jeep	Truck	Accuracy
Tank	16	0	0	0	94. 1%
POL	0	0	0	0	0%
Jeep	2	0	0	1	0%
Truck	0	0	0	1	50%

Table 4.11 Test Data Confusion Matrix for Momentum Method. Reads row by row left to right. Notice that two jeep feature vectors were classified as a tank, while the other was classified as a truck.

Class	Tank	POL	Jeep	Truck	Accuracy
Tank	17	0	0	0	100%
POL	0	0	0	0	0%
Jeep	2	0	0	1	0%
Truck	0	0	0	2	100%

Table 4.12 Test Data Confusion Matrix for Second Order Method. Reads row by row left to right. Notice that two jeep feature vectors were classified as a tank, while the other was classified as a truck.

Class	Tank	POL	Jeep	Truck	Accuracy
Tank	17	0	0	0	100%
POL	0	0	0	0	0%
Jeep	2	0	0	1	0%
Truck	0	0	0	2	100%

It should not be too surprising to observe that the second order and momentum methods perform slightly better than the gradient of steepest descent method, given the earlier results.

The gradient of steepest descent method will eventually reach the performance levels of the other two methods given more training iterations.

The Confusion Matrices above reinforce the results provided by Ruck [12]. The results, in all of the Test Data Confusion Matrices, confirm Ruck's original hypothesis [12]. The small number of training features (other than tanks) did not provide enough information for the network to properly segment the input decision space. However, even though the second order algorithm performed as well as the momentum method, it did not provide any improvements.

4.4. Summary

The first stage of validation was to show that the SO algorithm proved successful in solving the XOR problem. The proof basically duplicated, as well as verified the results found by Parker [8]. The SO approximation not only solved the XOR problem, but provided faster convergence on the average. The ability of the SO algorithm to classify feature vectors generated from doppler imagery, hinted that the algorithm could be used on other types of classification features. It was found that the momentum and second order methods slightly exceeded the performance of the gradient of steepest descent method. Furthermore, the second order and momentum methods performed on a comparable level.

A discussion of the general results of this chapter, and those in chapter 5 will be entertained in chapter 6, within the discussions section. The next chapter is devoted to the classification of features generated from Forward Looking Infrared Imagery.

5. Classification Results of Forward Looking Infrared Imagery

5.1. Introduction

The results of chapter 4 conclude that the second order minimization technique proved to be relatively successful. This chapter concerns various classifications of features generated from forward looking infrared (FLIR) imagery. As mentioned earlier in chapter two, other types of features, as well as moment invariants, will be considered for classification.

The next section of this chapter deals directly with the classification of those features generated for comparison with the Bayesian classifier. This classification effort will be based on target (TGT) and non-target (NT) recognition. The features selected for classification were the normalized versions of the blob length to width ratio, blob relative mean intensity, and blob standard deviation of the intensity (section 2.2.2).

The following section concerns the classification of the moment invariant feature vectors. The same comparisons drawn for the doppler imagery in chapter 4, will be used again in this section for the FLIR imagery.

5.2. Target and Non-Target Feature Classification

There were many feature vectors available for classification and approximately 75% of each class was used for training. Of the 819 feature vectors available for input, 615 were used for training; the others made up the testing data base.

5.2.1 Input Feature Data

Objects making up the TGT class consisted of tanks (TA), trucks (TR), APCs (AP), and jeeps (CJ). There were also several features generated from the combination of a tank and jeep (TC). The two targets were too close together to be resolved by the segmentation process. Table 5.1 breaks down the number of samples provided by each TGT, as well as providing the number of NT samples, for both training and testing.

Table 5.1 TGT and NT Sample Breakdown

Class	# Training Samples	# Testing Samples
TA	60	17
TR	80	25
AP	85	28
CJ	25	10
TC	15	8
TGT Total	265	88
NT	350	116

The raw features generated from the FLIR imagery, consisted of a wide range of values. In order to prevent the larger valued features from biasing the network, a normalization scheme was required. An attempt at computing a linear normalization scheme, placing all the data within the unit hypercube, failed to produce desired results. The network failed miserably when attempting to train on, and classify the feature vectors. This failure prompted an attempt at another normalization scheme. The training feature vectors were normalized to a 0 mean vector and a standard deviation vector of 1, as described in section 2.2.1. This normalization scheme proved to be much more successful then the previous scheme and the results are provided below in section 5.2.3.

The notion that the first normalization scheme failed and the second was somewhat successful, may provide some insight as the function of neural net classifiers. It appears that the network, not only cares about the magnitude of each vector, but in addition cares a great deal about the angle between vectors. It appears from this argument that the network is functioning as a nearest neighbor classifier.

5.2.2. Network Architecture and Learning Parameters

The network architecture used for classifying this set of feature vectors is described in Table 5.2. Table 5.3 describes the network training data for all three minimization techniques.

Table 5.2 Network Architecture Data

Number of Features	3
Layer One Nodes	50
Layer Two Nodes	20
Number of Classes	2

Table 5.3 Network Training Data

Parameter	Gradient Method	Momentum Method	SO Method
a_1	0.25	0.25	0.25
a_2	0.0	0.0	0.0
a_3	0.0	0.0	0.0
a_4	1.0	0.3	0.3
a_5	0.0	0.0	0.1
Number of Iterations	200,000	200,000	200,000
Data Output Interval	4,000	4,000	4,000

The weights and thresholds were initialized to values within the interval (-0.45, 0.45) using a uniform random number generator.

5.2.3. Classification Results

This portion of the text provides the classification results of the neural net classifier. The instantaneous classification accuracy versus the number of iterations is provided below. Due to the large number of input training vectors and the enormous number of training iterations required, the results were not averaged. The log error is also considered, along with the tally on the TGT and NT individual accuracies. Again, comparisons between the different methods will be drawn for both the training data and testing data. In addition, the performance results of the Bayesian classifier are presented in this section as well.

5.2.3.1 Instantaneous Classification Accuracy

For the neural net classifier, the same criterion as the one used in section 4.4.3.1 is used again here. The desired output node must fire greater than or equal to 0.8, while the remaining nodes fire at 0.2 or less. By using this criterion, the neural network classifier has somewhat of a disadvantage, when compared to the Bayesian classifier. Recall that the Bayesian classifier uses maximum a posteriori decision criterion. This is a more lenient criterion than the one placed on the neural network classifier. Appendix E considers a comparison between the two classifiers, given a more lenient criterion placed on the neural net classifier.

The results of each method follows, with the training data first, followed by the testing data. The results from the Bayesian classifier are provided next and comparisons between the two classifiers conclude this subsection. Figures 5.1, 5.2, and 5.3 consider the training data, while Figs. 5.4, 5.5, and 5.6 consider the test data.

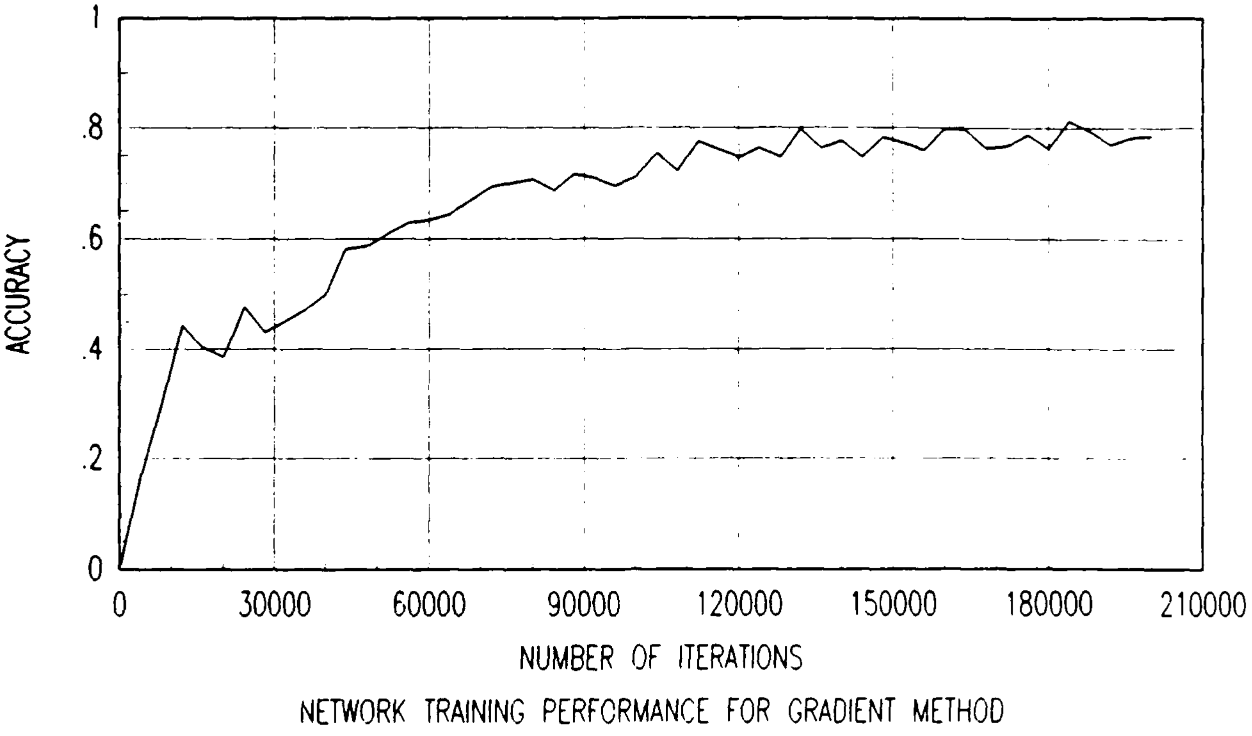


Figure 5.1 In comparison with Figs. 5.2 and 5.3, the gradient method is slower in convergence, reaching approximately 82% classification accuracy.

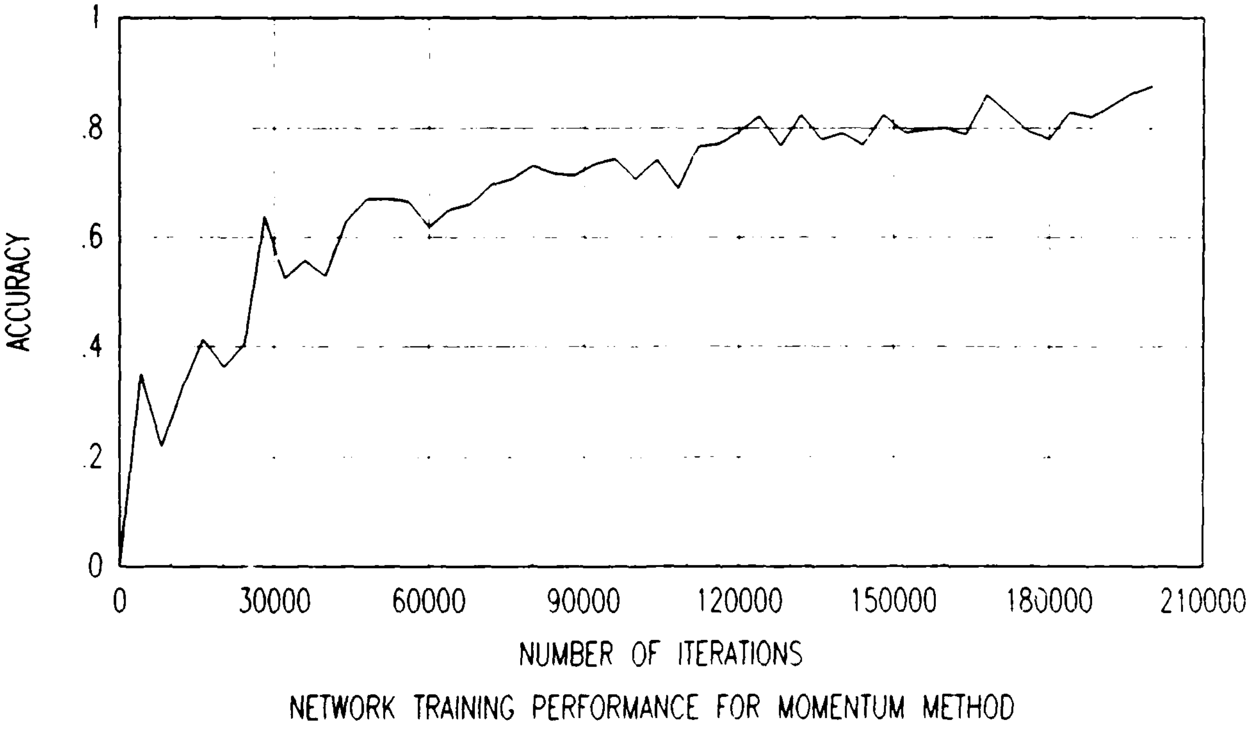


Figure 5.2 The network achieved over 87% accuracy on training data.

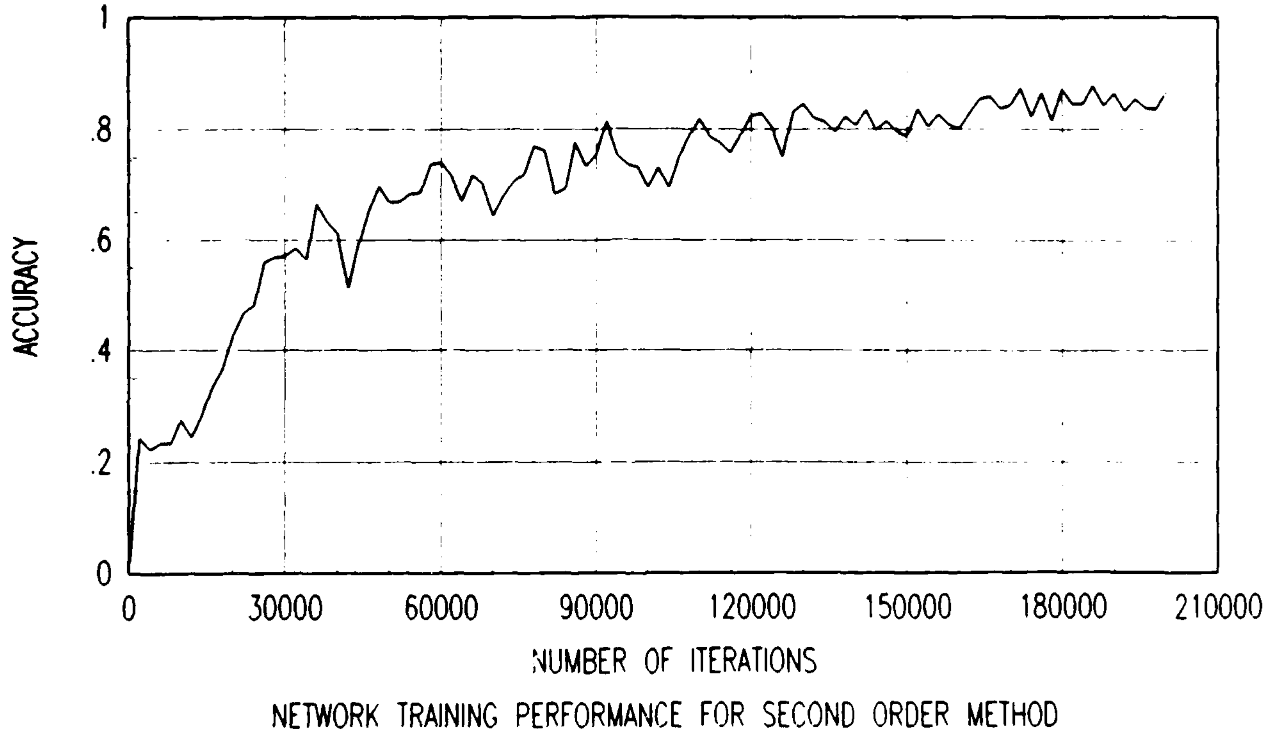


Figure 5.3 The network achieved over 87% accuracy on training data. The second order method reached this accuracy faster than the other two methods

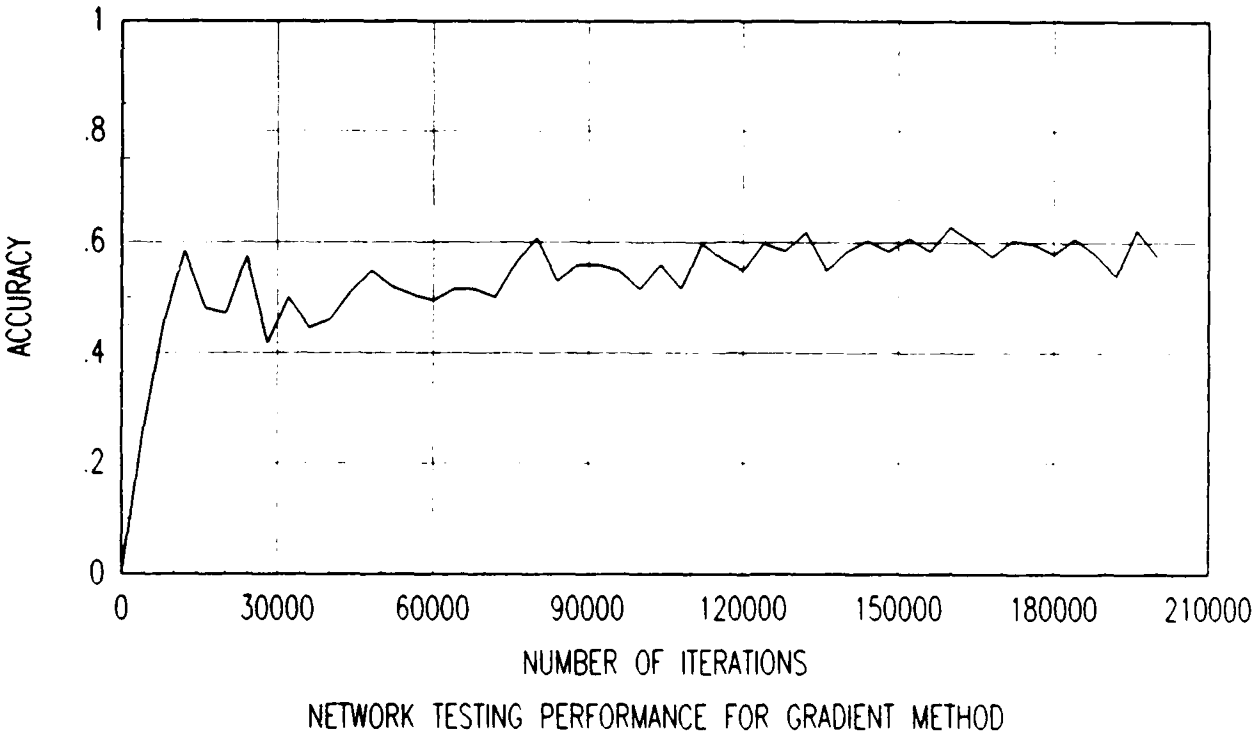


Figure 5.4 The test data accuracy was poor, reaching only 62%.

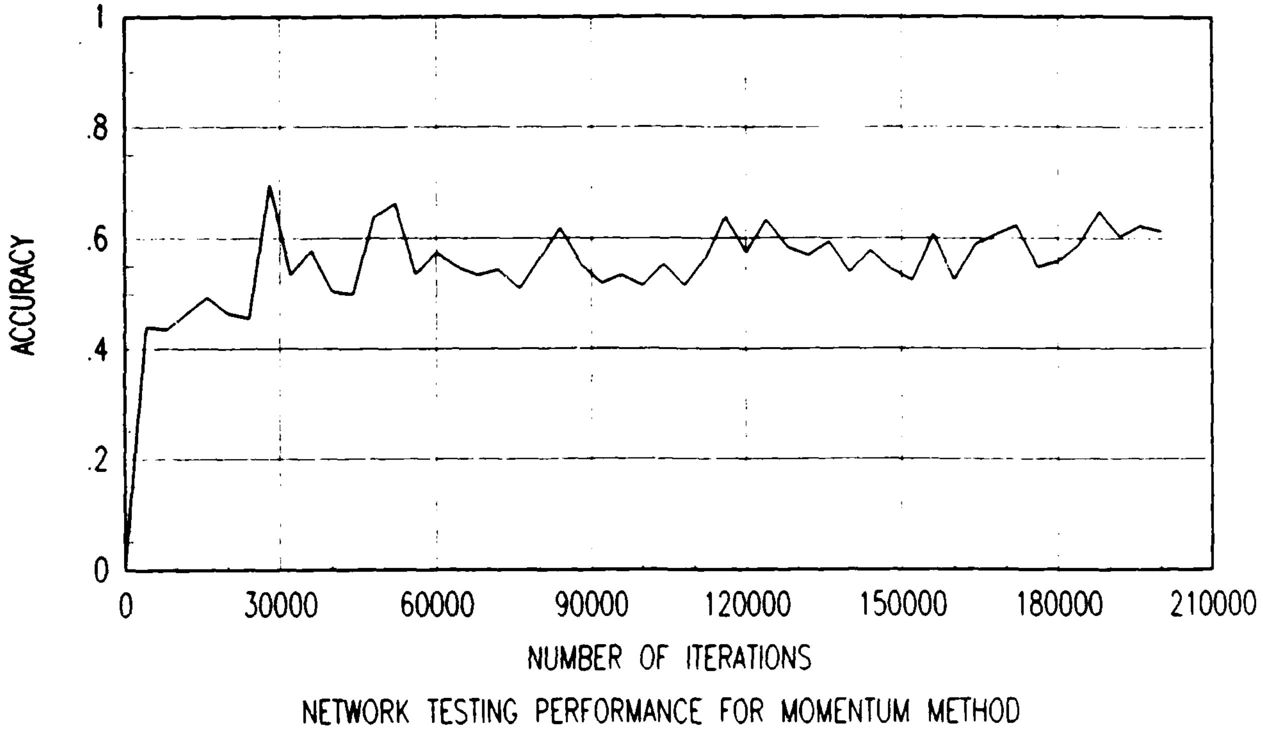


Figure 5.5 The network averaged roughly 60% accuracy over the last 30,000 iterations. Not much improvement over the gradient method.

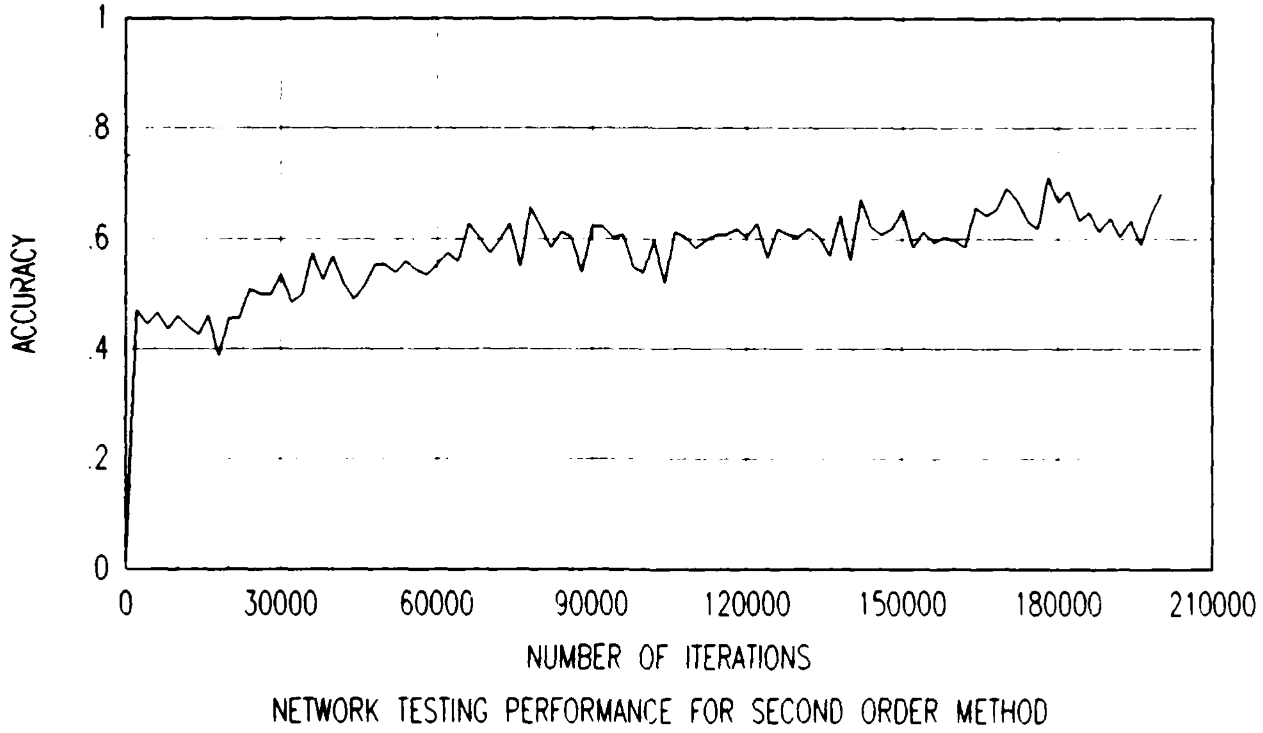


Figure 5.6 The network averaged over 65% accuracy over the last 30,000 iterations, revealing a distinct improvement.

Again, the momentum and second order methods continue to exceed the performances of the gradient of steepest descent method, but only slightly. The differences between the momentum and second order techniques, again are minimal. However, close examination of the data reveals that the second order method begins to exceed the performance of momentum method at 30,000 iterations for the training data. The second order method also has a distinct advantage over the momentum method during the last 30,000 iterations. These same observations are carried over into the test data results once the network reaches 60,000 iterations. Although the results as a whole were not terribly exciting, the fact that the second order method provided better accuracy, in fewer number of iterations is significant. However, the network performance has not been averaged and these results should not be taken out of context. More testing is required, such that the network performance may be averaged. In addition, it would be desirable to increase the number of features.

5.2.3.2. Average Total Output Error

The same criterion for determining the average output error in section 4.4.3.2 is used here. The average total output error was measured only for the training data of all three methods. The training error is given so that the decreasing trends may be observed and verified.

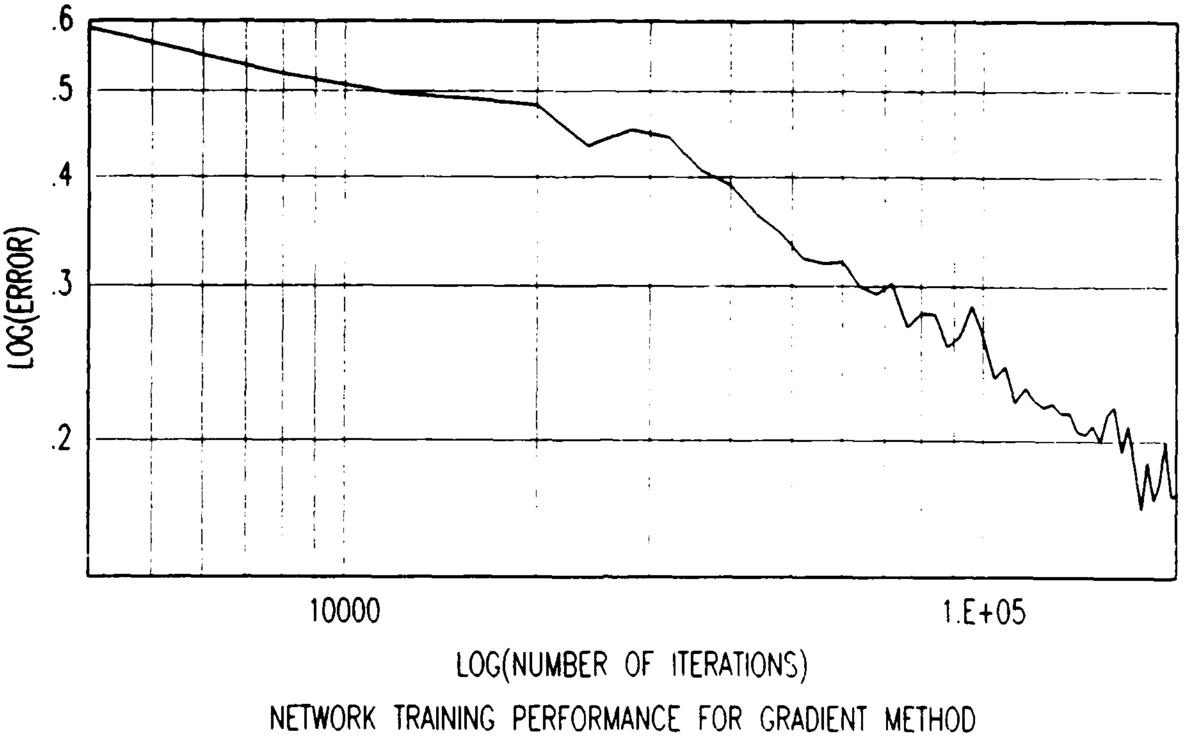


Figure 5.7 Notice that the error plot is much smoother than in Figs 5.8 and 5.9.

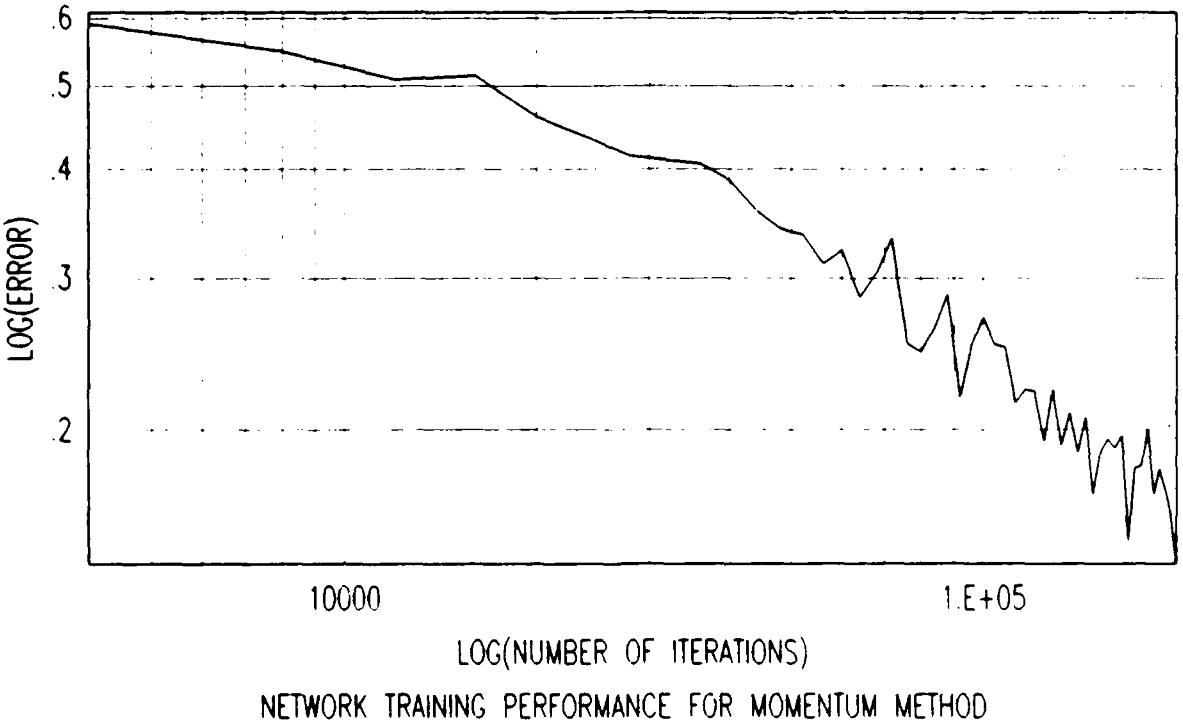


Figure 5.8 Notice the smooth initial reduction. Although rough in the latter training stages, the error continues to decrease.

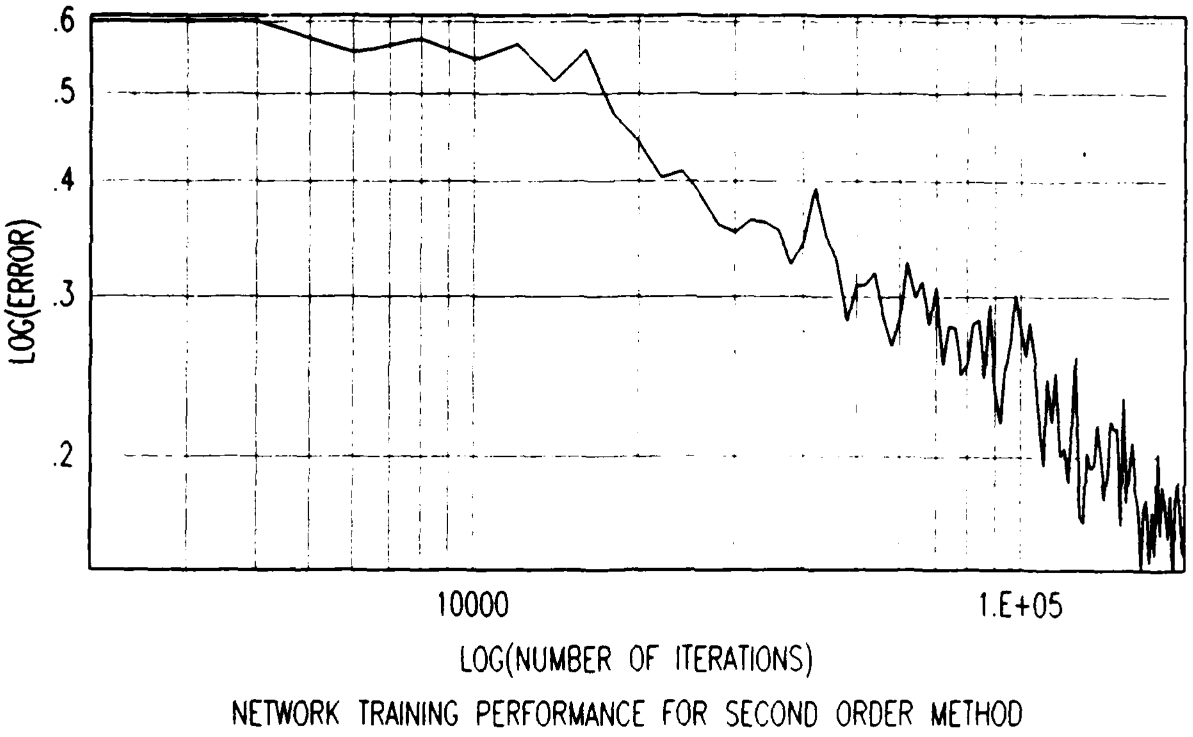


Figure 5.9 Although rough, the error continues to decrease over the course of training.

Here again, the momentum and second order methods are slightly more successful than the gradient of steepest descent method. As noted earlier, there is still minimal differences between the momentum and second order methods. The error was extremely unstable, but decreasing none the less. Averaging the data would have smoothed the results quite a bit. Again, further testing is required.

Notice that the roughness, in later training iterations, appears more significant with the second order method, than either of the other two methods. In turn, the momentum method is rougher than the gradient method. Again, part of this would be suppressed by averaging. However, the results may further suggest another reason. The choice of learning parameters may not be suitable for decreasing the error in an efficient manner. This suggests that the learning parameters may also have to updated at various stages in training, to enhance learning.

5.2.3.3. Neural Net Classifier versus Bayesian Classifier

The following tabulated results demonstrate the comparisons of classification accuracy between the statistical Bayesian Classifier and the neural net classifier. The results are broken down in Table 5.4. Each method of the neural net classifier is considered, after the network had achieved 200,000 training iterations. The results provided below are just typical instances of each method; the results have not been averaged. Again, keep in mind that the criteria used for each classifier is different, with neural net classification criterion being much more stringent. Appendix E demonstrates a more comparable criterion.

Table 5.4 Classification Accuracy of Neural Net Classifiers versus the Bayesian Classifier. (1) Gradient Method, (2) Momentum Method, (3) Second Order Method, (4) Bayesian.

Accuracy				
Class	(1)	(2)	(3)	(4)
TGT Training	73.6%	81.4%	82.8%	80.3%
NT Training	83.1%	87.0%	86.6%	70.6%
TGT Testing	57.9%	61.4%	64.8%	76.9%
NT Testing	57.8%	62.2%	64.7%	74.7%

The neural net classifier exceeds the performance of the Bayesian on the training data and the roles are reversed for the test data. It is believed, that by increasing the number of features, this may provide more information to the classifiers, and especially to the neural net classifier. The neural net classifier learns by example, more examples, then more information is provided to the net. In other words, the net learns more about its environment. Thus, the net would have more of an opportunity to extract the essence of a particular object.

5.3. Moment Invariant Feature Classification

This section concerns the classification of the moment invariant features. The following subsection describes the input training data. Next, the network architecture is discussed, followed by the classification results.

5.3.1. Input Feature Data

The classes considered for this study consisted of tanks (TA), Trucks (TR), and armored personnel carriers (AP). Initially, there were targets generated from two fields of view, narrow and wide. The narrow field of view consisted of 3.43 degrees in the horizontal and 2.57 in the vertical. The wide field of view consisted of 10.32 degrees in the horizontal and 7.74 degrees in the vertical. For reasons to be explained later, the narrow field of view objects were used exclusively. There were 104 feature vectors, of which 75% were used for training, while the remainder were used for testing. Table 5.5 breaks down the number of samples for each category for the narrow field of view targets.

Table 5.5 Target Data Base

Class	# Training Samples	# Test Samples
TA	25	7
TR	25	5
AP	25	17

There was a relatively even distribution of vectors describing each target, allowing an even distribution of the training data, as opposed to the breakdown listed in Table 4.4. In general, the more examples of an object the network has to train on, the better chance the network has of learning that object.

5.3.2. Network Architecture and Learning Parameters

The multilayer perceptron consists of 3 layers, which will receive 36 input features and output 4 classes. Table 5.6 describes the network architecture, while Table 5.7 defines the predetermined learning parameters.

Table 5.6 Network Architecture Data

Number of Features	36
Layer one nodes	27
Layer One Nodes	9
Number of Classes	3

Table 5.7 Network Training Data

Parameter	Gradient Method	Momentum Method	SO Method
a_1	0.1	0.1	0.1
a_2	0.0	0.0	0.0
a_3	0.0	0.0	0.0
a_4	1.0	0.1	0.1
a_5	0.0	0.0	0.1
Number of Iterations	20,000	20,000	20,000
Data Output Interval	400	400	400

Many combinations of learning parameters were tried, but the search was not exhaustive. The learning parameters listed in the above table provided the best results.

5.3.3. Classification Results

The same comparisons as those made in section 4.4.3.1 will again be drawn in this section for the FLIR imagery features. The average classification accuracy, as well as log error plots are considered. Again, results of all three minimization methods will be compared.

5.3.3.1. Average Classification Accuracy

Here again, the same criterion used for classification in section 5.2.3.1, is used for these accuracy results. Initially, both wide and narrow field of views were used for classification. However classification accuracies never exceeded 63% on the training data. The images generated from the wide field of view produced poor target resolution. Targets were not distinguishable from non-target blobs, much less from one another, to the human observer. The results of removing those features segmented, from the wide field of view images, from the target data base are displayed below. Figures 5.10, 5.11 and 5.12 display the average training accuracy.

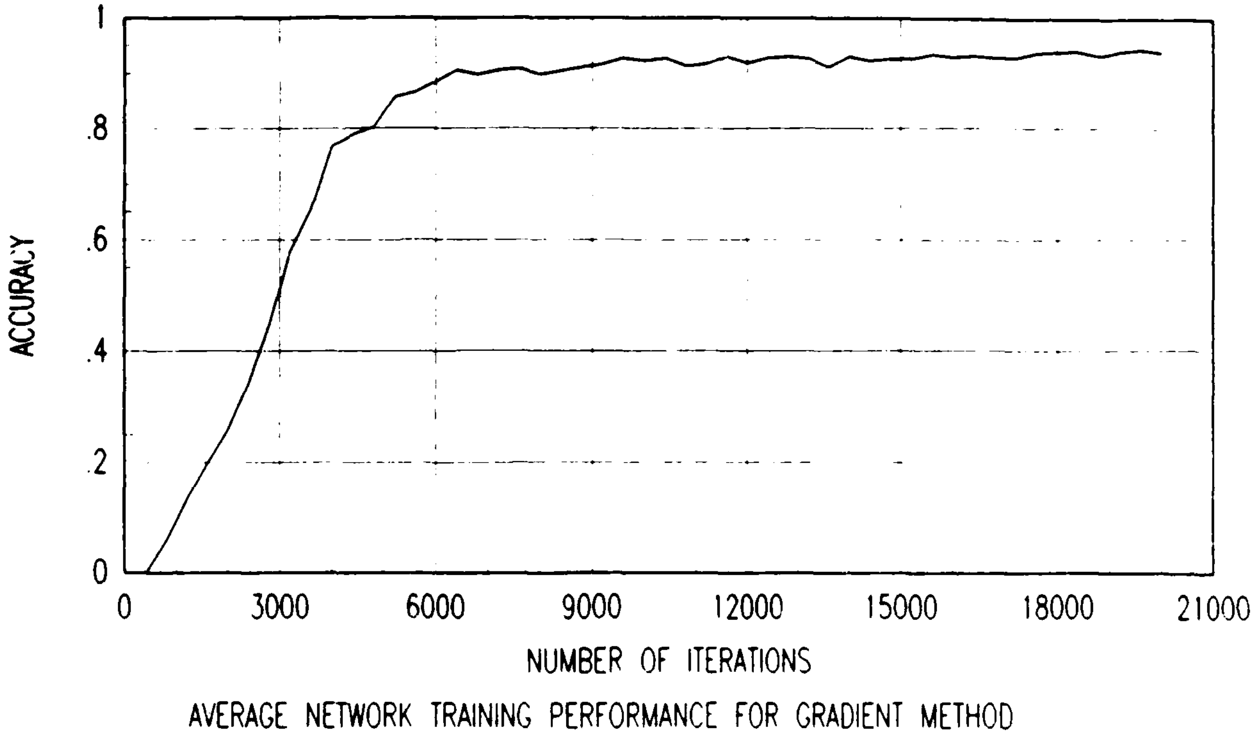


Figure 5.10 The network achieves accuracies just under 95%.

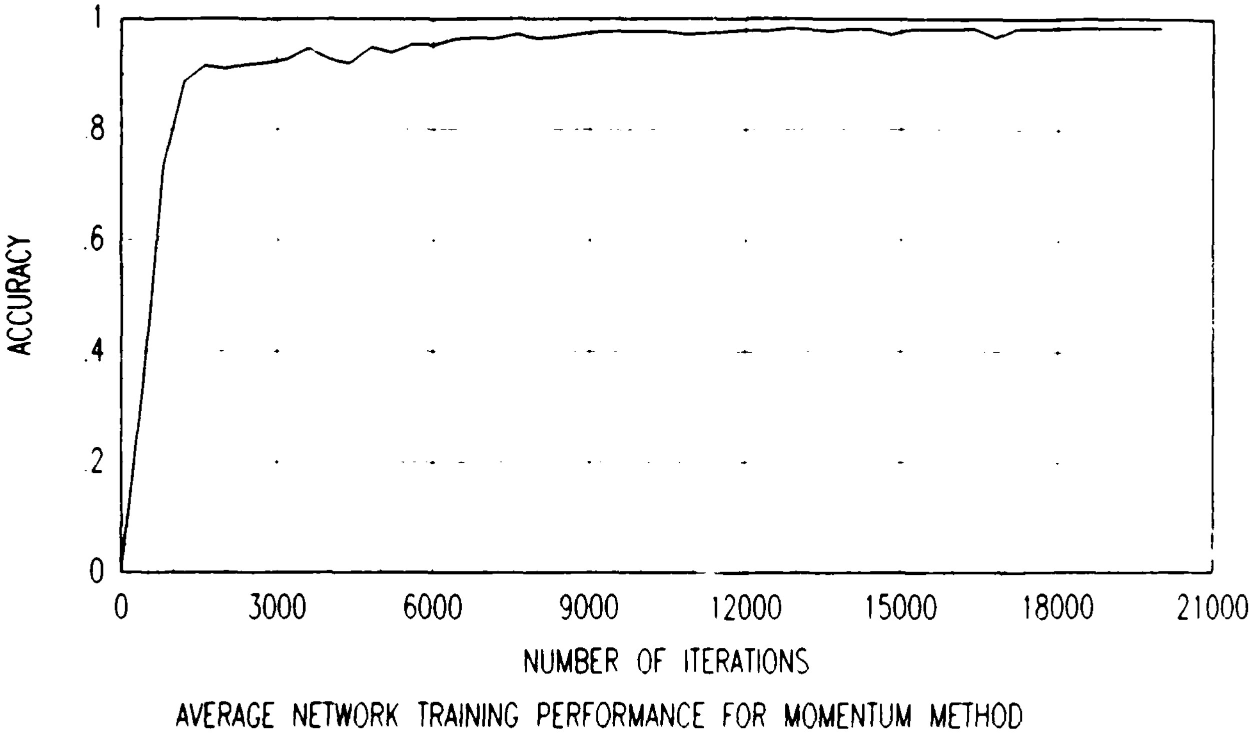


Figure 5.11 The network achieves accuracies of 98%.

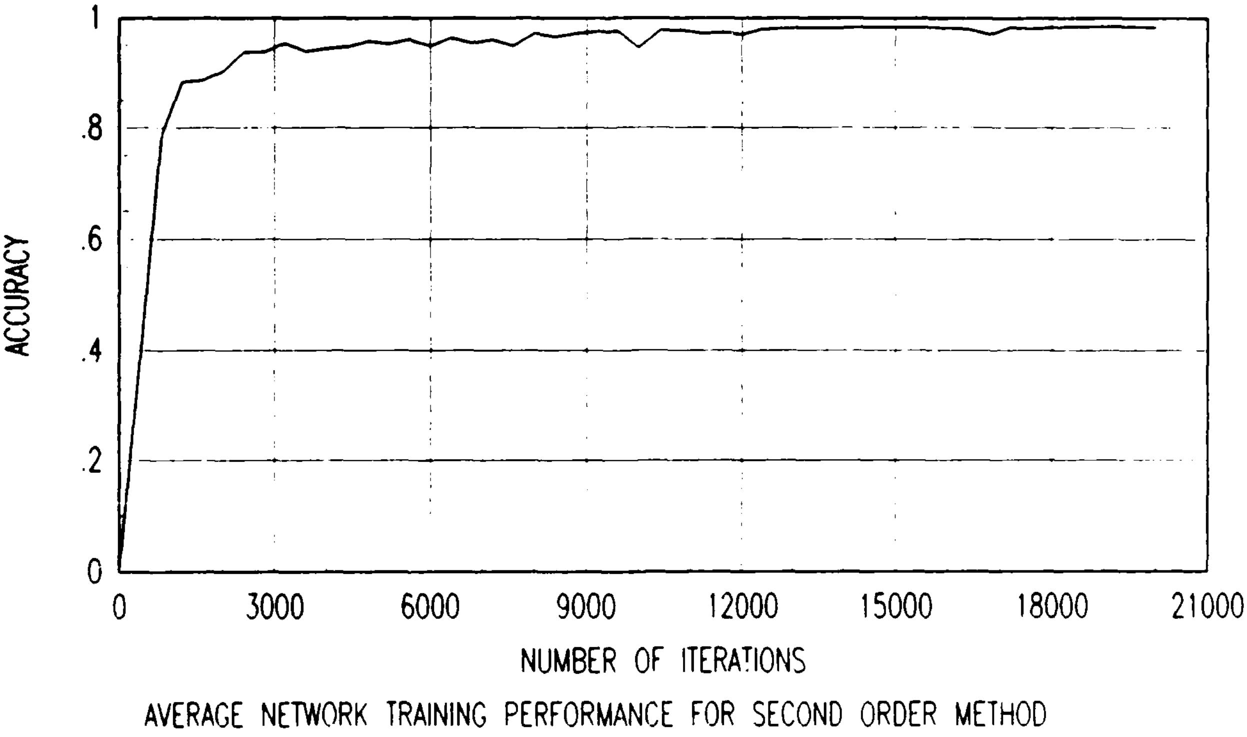


Figure 5.12 The network achieves accuracies of 98%.

Below, Fig. 5.13 displays the first 4000 iterations for each method on the same graph for a better comparison.

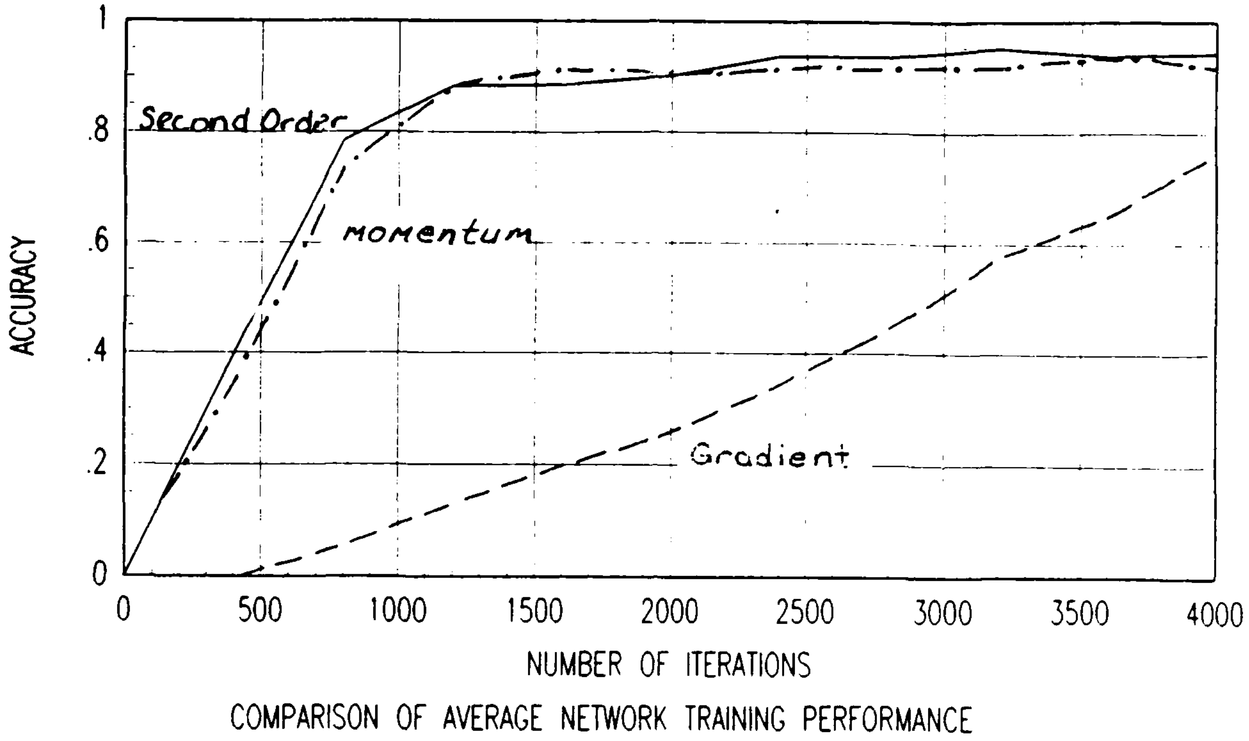


Figure 5.13 The momentum and second order methods clearly exceed the performance of the gradient of steepest descent. In addition, notice that the second order method on the average converges somewhat faster than the momentum method.

Figures 5.14, 5.15 and 5.16 represent the test data accuracies of each method.

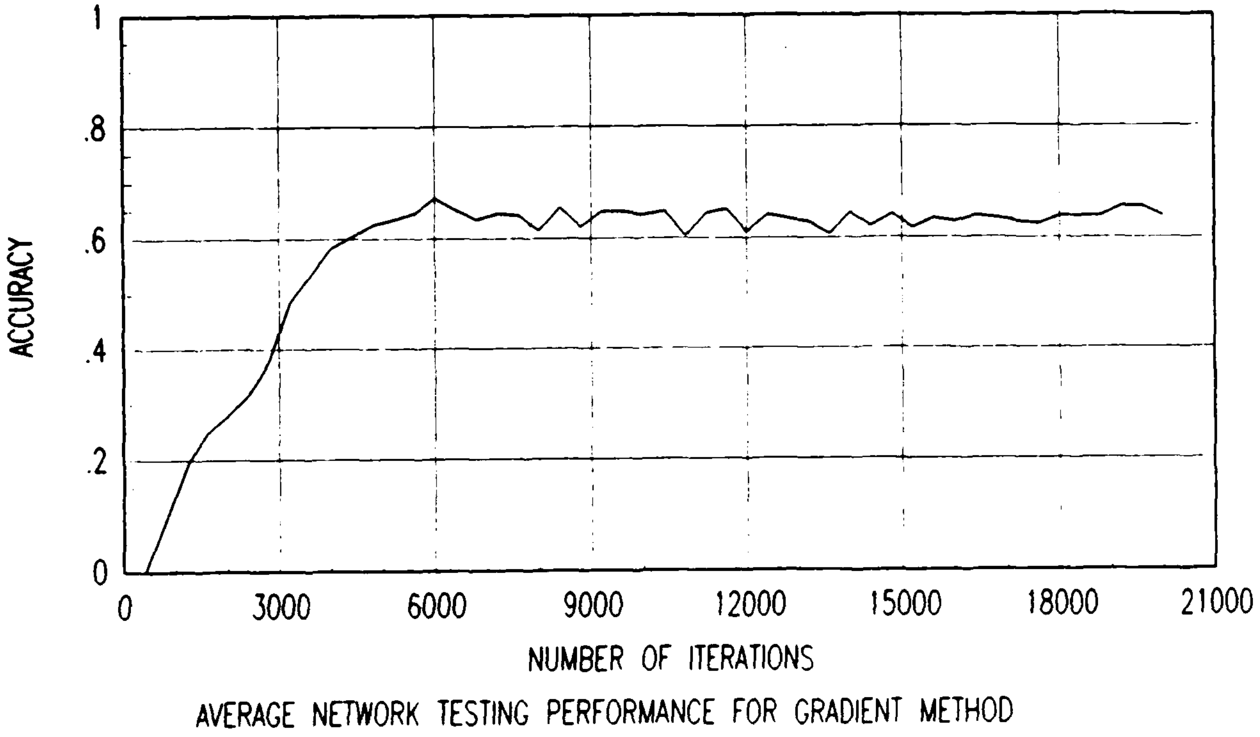


Figure 5.14 The network achieves close to 65% accuracy.

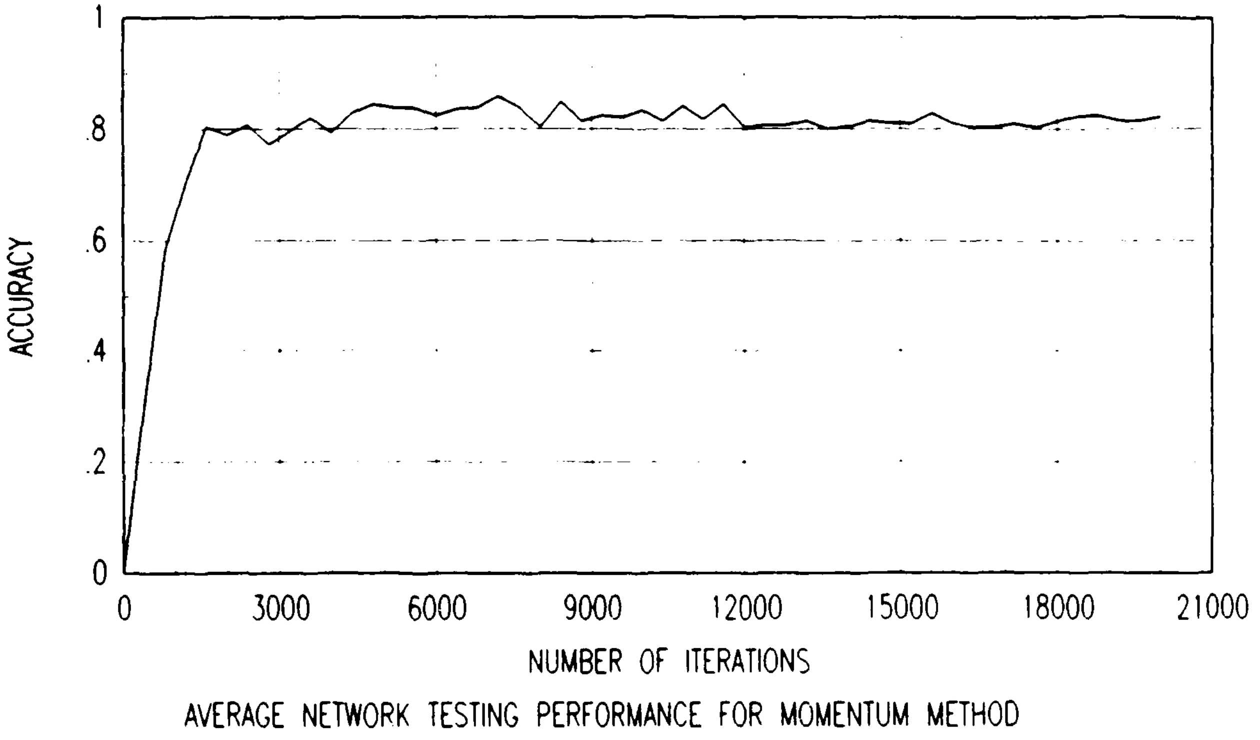


Figure 5.15 The network achieves 85% accuracy. Notice how the accuracy deteriorates over continuous training.

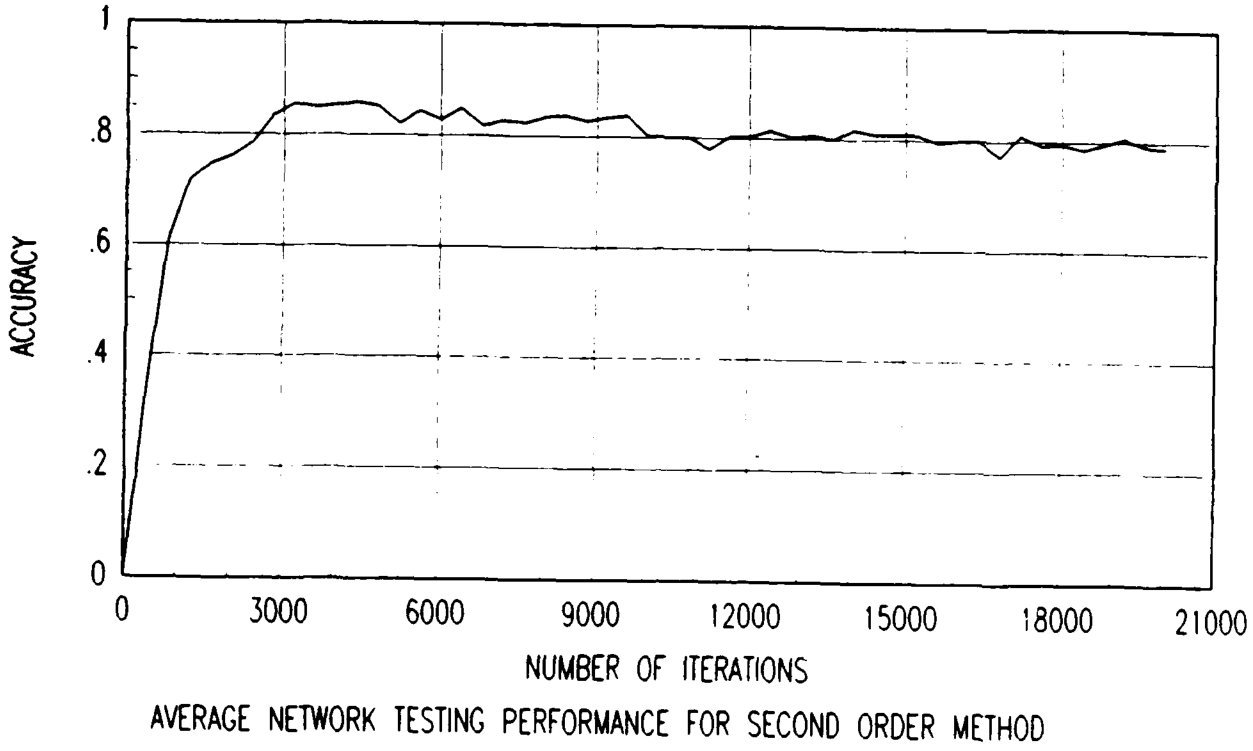


Figure 5.16 The network achieves 85% accuracy. Notice how the accuracy deteriorates over continuous training.

Figure 5.17 displays all three plots over the first 4,000 iterations for the test data.

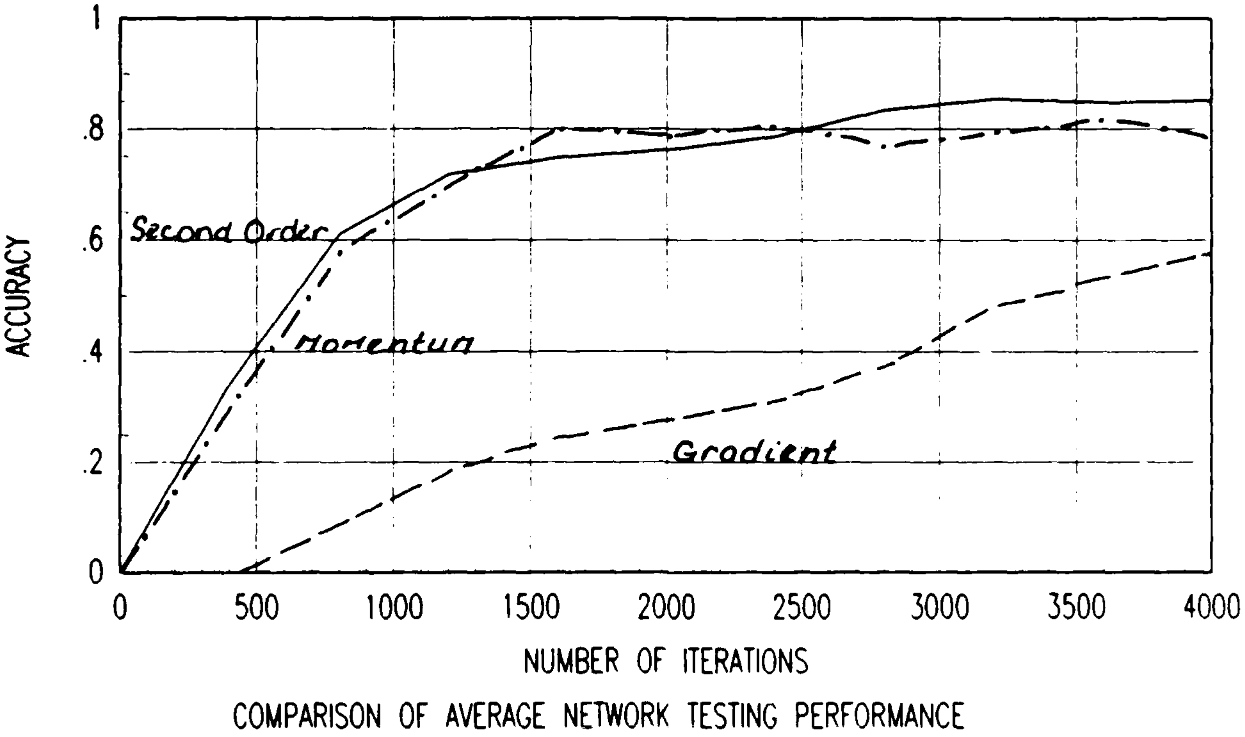


Figure 5.17 The momentum and second order methods clearly exceed the performance of the gradient of steepest descent. In addition, notice that the second order method performs a little better than the momentum method.

Once again, the gradient of steepest descent has failed to perform on a comparable level with the other two minimization techniques. In the case of the doppler imagery, there was not the significant difference between the different techniques, as observed here with the FLIR imagery feature vectors. A closer look at the accuracy plots, reveals that the SO method initially converges a little faster than the first order momentum method, as shown in Fig. 5.13 and reinforced in Fig.5.17.

In Figs. 5.15 and 5.16, observe that after about 1,200 or so training iterations the test data accuracy begins to deteriorate. This phenomenon may be explained more clearly by analyzing the average total output error.

5.3.3. Average Total Output Error

Again, the average total output error is defined exactly the way it was in section 4.4.3.2. However, the total output error was averaged over 75 feature vectors for the training set and averaged over 29 vectors for the test set. The log error will be plotted for the training data and test data.

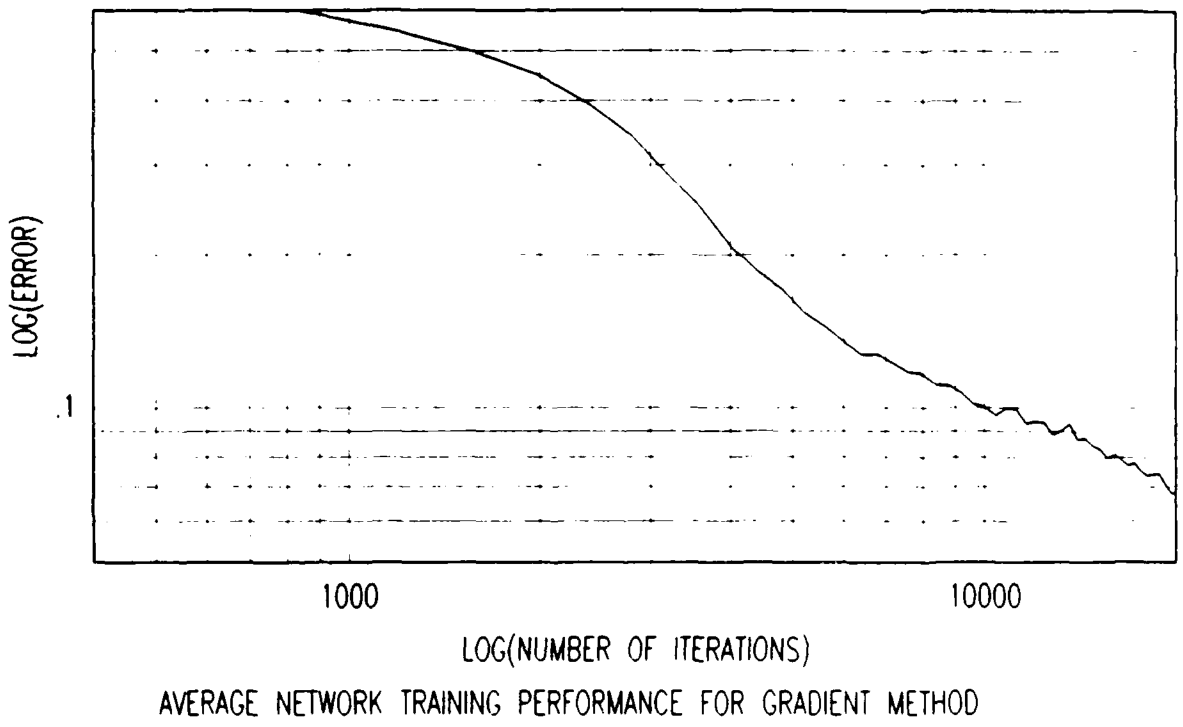
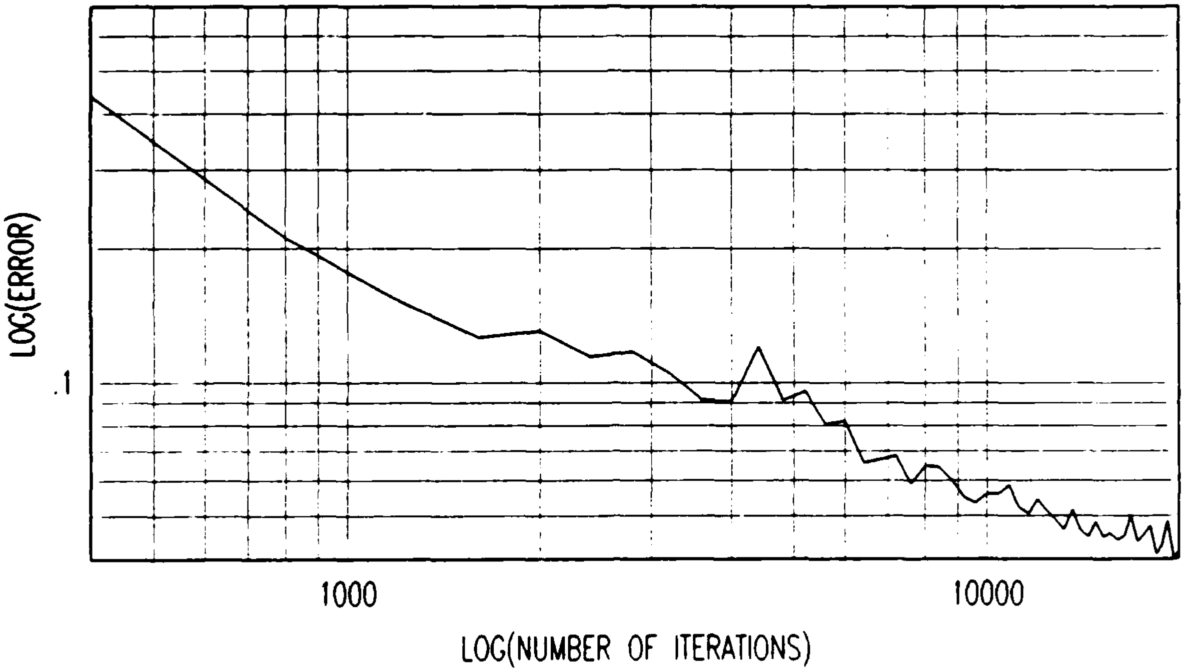
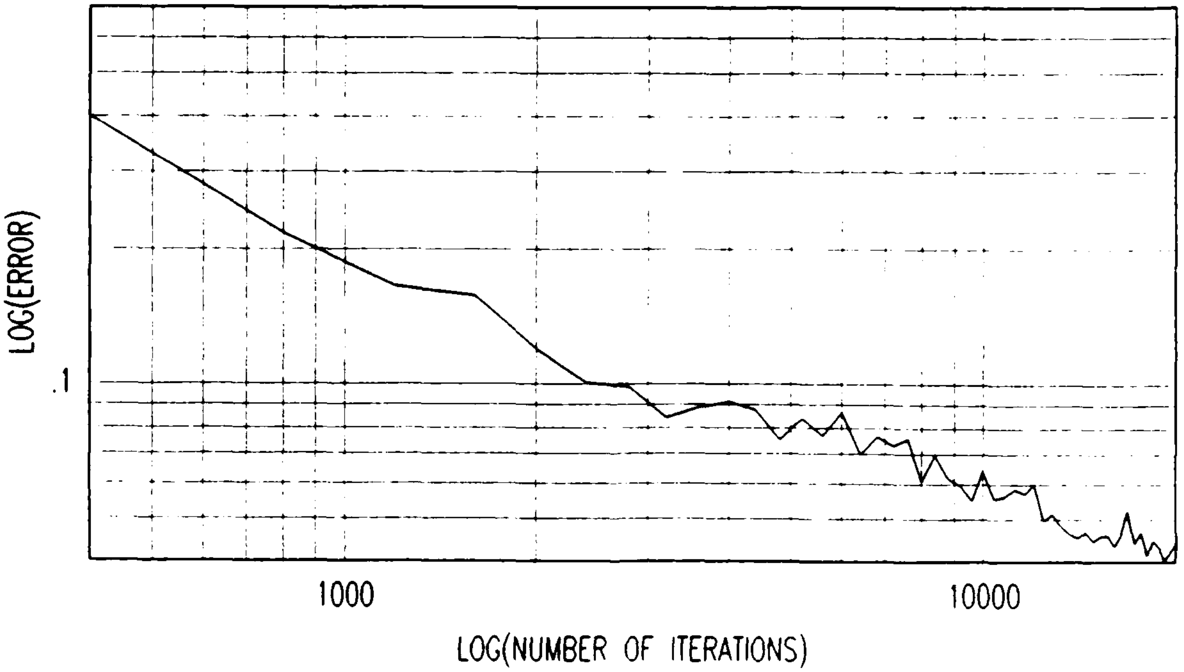


Figure 5.18 Notice the smooth descent over all training iterations.



AVERAGE NETWORK TRAINING PERFORMANCE FOR MOMENTUM METHOD

Figure 5.19 Notice the reduction in error over the gradient method.



AVERAGE NETWORK TRAINING PERFORMANCE FOR SECOND ORDER METHOD

Figure 5.20 Note the minimal difference between the momentum and second order methods.

Notice, that in each graph the log of the error is basically a smooth decreasing curve to about 1,200 training iterations, after which the curve becomes very unstable. The average output error for the test data, will provide a bit more insight to this peculiarity. The average output error for the test data will also support the information contained in Figs. 5.15 and 5.16.

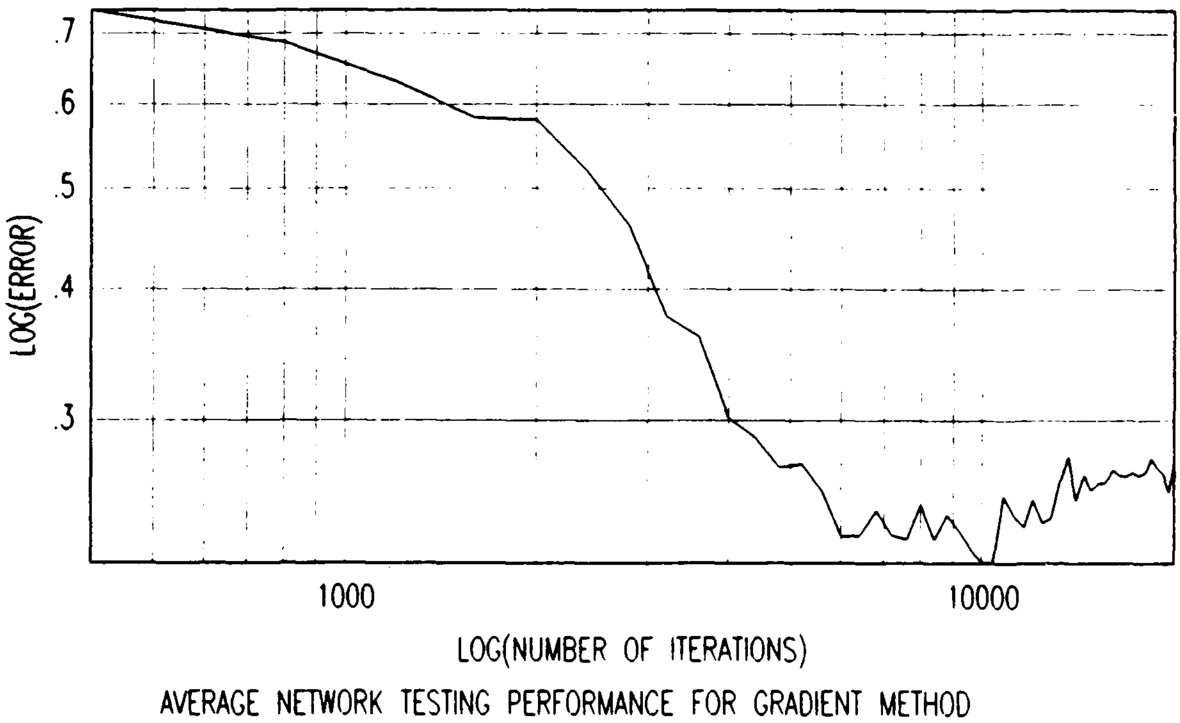


Figure 5.21 Notice the increase in error at roughly 10,000 iterations.

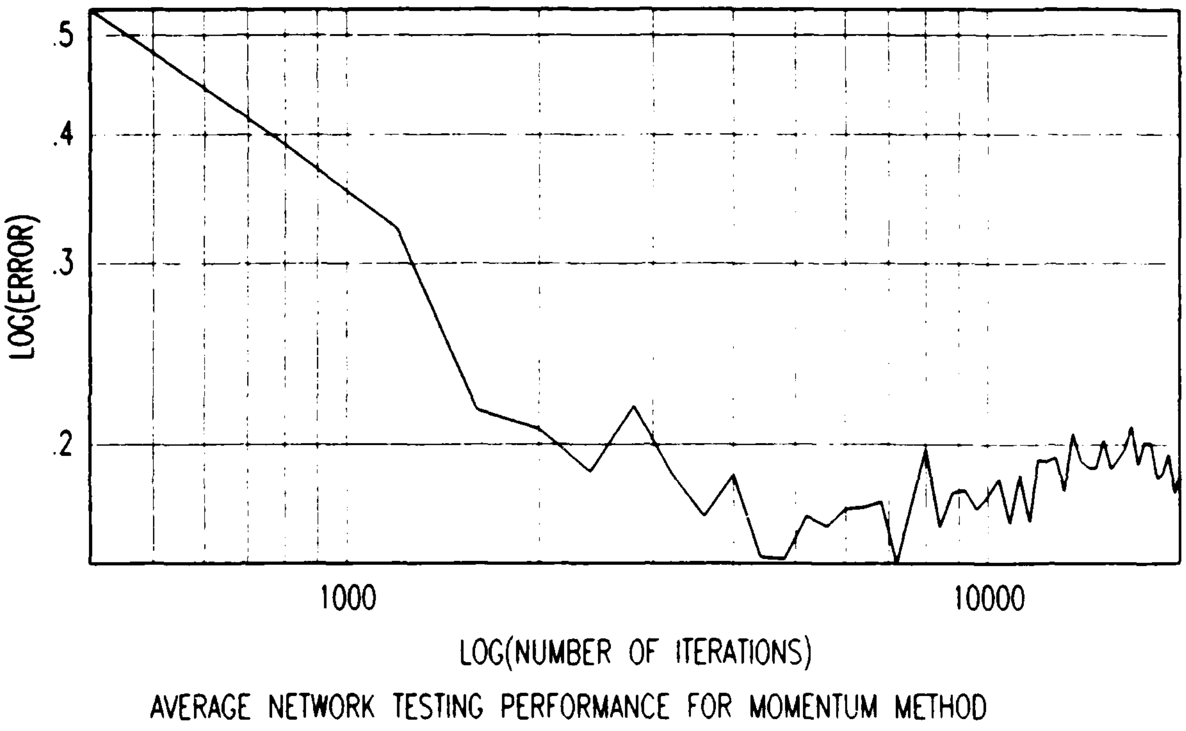


Figure 5.22 Notice the increase in error at roughly 1,350 iterations.

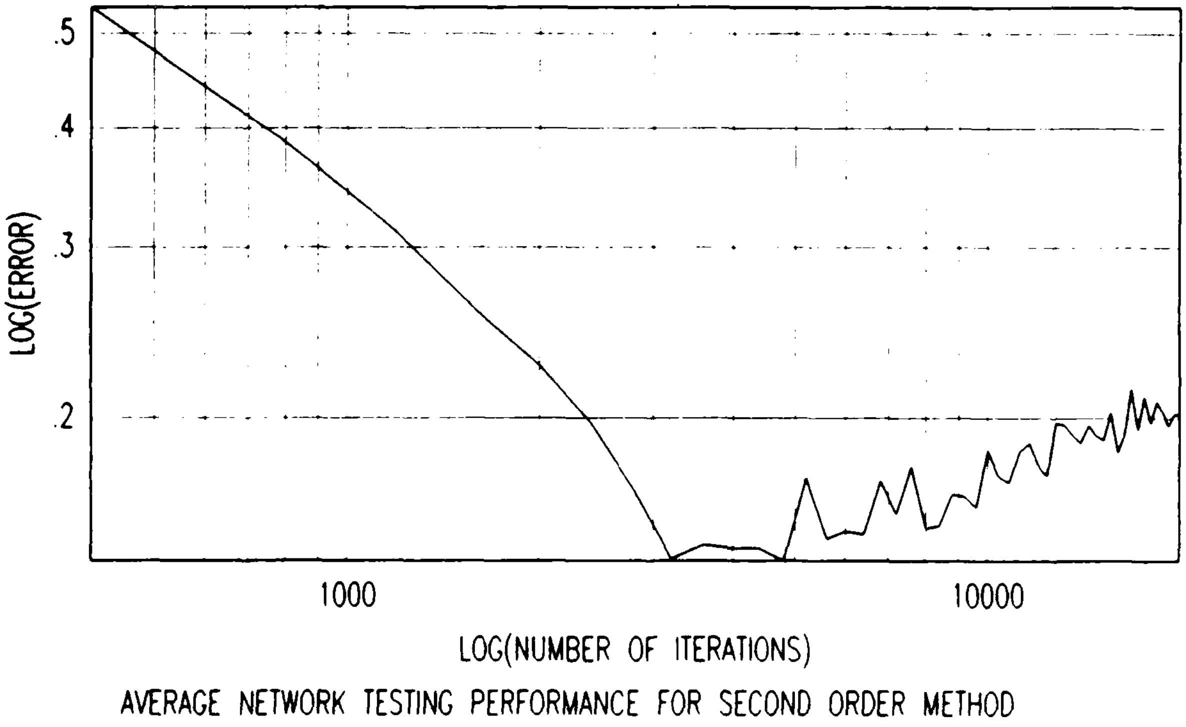


Figure 5.23 Notice the increase in error at roughly 1,200 iterations.

In each error plot, the error is decreased, to some minimum and then abruptly begins to increase. This phenomenon agrees with observations made in Figs. 5.15 and 5.16.

Considering Figs. 5.18 and 5.19, it appears the network is learning and converging on the optimal weight values which minimize the error surface. However, as the feature vectors are presented to the network, over and over, a point is reached when the network attempts to find an exact solution to the optimum weight values. If the region of the minimum is a relatively flat region, there may be many solutions. Therefore, the net begins to memorize and the network loses its ability to generalize and abstract the essence of the target. This may explain why the average network performance begins to degrade on the test data. The network expects data which closely resembles the training data. When it does not see this resemblance, then it makes an inaccurate decision.

5.4. Summary

Classification of the target and non-target features using the neural net classifier exceeded the performance of the Bayesian classifier for the training data. But, the Bayesian classifier performed better on the testing data. However, by applying a more lenient classification accuracy on the neural net classifier it is believed that the network performance will be improved. Appendix F provides the results when applying a more lenient criterion to the neural net.

Classification of the moment invariant features proved quite successful. The momentum and second order methods clearly exceeded the performance of the gradient method of steepest descent. Classification accuracies near perfection for the training data were measured, while accuracies of 85% were achieved for the test data. Unfortunately, there was not a big difference between the performance of the momentum method and second order method, other than the second order method providing a slight improvement in convergence time.

Chapter 6 entertains a general discussion on the results of all three minimization techniques used in chapters 4 and 5. Aside from these discussions, recommendations and conclusions are provided to conclude the thesis effort.

6. Discussions, Recommendations and Conclusions

The following sections provide the closing remarks concerning the results of this thesis effort. The chapter begins with a discussion on the general results and findings provided in chapters 4 and 5. Furthermore, the areas of further study in pattern classification using neural networks, specific to this research, are considered in section 6.2. The conclusion discusses the contributions of this effort in the field of tactical target recognition using neural network classifiers.

6.1. Discussions

The findings obtained in chapters 4 and 5 provide some very interesting results. For example, with three different sets of input feature vectors, it was shown that there were distinct disadvantages of applying the gradient method of steepest descent as a minimization technique. In every case studied the momentum and second order approximation methods exceeded the performances of the gradient of steepest descent technique. There are several reasons for this and a few are discussed below in the paragraphs to follow.

When applying the steepest descent method, recall that the weights are being updated in small constant proportional increments of the partial derivative of the performance surface. If the minimum of the error surface lies in a relatively flat hyperplane, the reduction in error will be very slow. In addition, the existence of local minima could also pose a significant threat to convergence. Another reason may lie in the fact that the gradient (the vector of partial derivatives) may not point in the direction of the global minimum. In weight hyperspace, the partial derivative of the error surface (the slope) in one weight direction may be far greater than the partial in yet another weight direction. Hence, the gradient would not necessarily point to the global minimum and the weight update may be of little consequence in convergence. Thus, these results confirm and reinforce the ideas and concepts of many researchers throughout the literature.

However, there is disagreement by many in the field of neural networks, as to the best way to accelerate convergence. A seemingly controversial means of acceleration is the momentum term. Again, in all applications in this study, the momentum method clearly exceeded the performances of the steepest descent method. One reason for this, is the additional amount of information the momentum method provides to the network concerning the error surface. With this method, the network is allowed to look back in time by one time step. This allows the network to add the so-called momentum term. This momentum term is simply a weighted version of the weight update from the previous time step, allowing convergence to continue in the same direction as the previous step. If the current update has the same sign as the previous update then the convergence towards the minimum is accelerated. If the signs are opposite then the update is small; it is hoped that this will prevent the network from over shooting the minimum. However, this may not be the case and the algorithm may also still be susceptible to local minima. Never the less, in general, the momentum term has the effect of adding a quadratic term to the minimum of the error surface. This term performs an average of the current and previous updates and the result is a smoothing of the error surface. This study shows the momentum method to be quite effective, when compared to the steepest descent method.

The second order approximation to Newton's method proved just as successful as the momentum method and in some instances slightly accelerated convergence. The basic concept behind accelerating convergence is to provide the network with as much information as possible about the ever changing error surface. In doing so, the decisions made by the network are made faster, more decisive and accurate. In using second derivative information, it is desired to gain new information. For instance, using first order techniques as those described above, all the information about the past training is stored the positional values of the weights. Second order methods store information about the local shape of the error surface, as well as maintaining the positional information. In addition, the algorithm used in this study, applies time derivative information. With each new input presented to the network, the performance surface changes. Therefore, when considering each time step, the performance surface is changing with time. The time derivatives of signals propagating through the network, provide the network information regarding how the performance surface changes with time. Any acceleration in convergence over the other two methods (within this study) has to be attributed to this added information this algorithm is providing the network.

The reason there was not much difference between the momentum and second order methods is not well understood. Perhaps the very fact that the algorithm used in this study, is merely an approximation to the more powerful Newton's method provides the answer. After all, the actual implementation of this approximation provided an additional term to the already existing momentum method within the algorithm (see Eq. 2.8). This additional term contained all of the second derivative information, as well as providing the time derivative information. The following section provides recommendations for areas of further study, to include techniques which may provide more information than the momentum method, and thus accelerate convergence.

6.2. Recommendations

The second order algorithm approximating Newton's method provided some promise for pattern classification. However, from the results of chapters 4 and 5, the second order algorithm provided very little improvement over the momentum method, other than a slight increase in convergence. Therefore, further studies are required for improving the performance of neural net classifiers.

1. In chapter 2, it was shown that Parker [8] derived an approximation to the powerful iterative technique known as Newton's method. More specifically, the algorithm approximates the following second order Newton's method:

$$\frac{\partial w}{\partial t} = \left[\mathrm{avg}\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \right]^{-1} \cdot \mu \cdot \frac{\partial s}{\partial w}.$$

Many problems arise when attempting to solve this linear differential equation. First of all the number of operations becomes a cubic function of the number of weights (n) in the network. These are the number of operations required to invert the matrix. However, as shown in appendix D, the above matrix has a singularity and therefore is not invertible. For these reasons and more, explains why an iterative approach was used to approximate the above linear differential equation. The approximation basically concerned the splitting of the matrix as show in appendix A. There are many ways to approximate the above equation by iterative techniques.

Aside from the approximation to Newton's method applied in this study, other approximations should be explored. One such approximation may apply the Jacobian method for splitting a matrix, in lieu of using the identity matrix. Another method may employ a Gauss-Seidal method. Still, another method, which may prove to be even more powerful, is the successive overrelaxation method which is a generalized version of the Gauss-Seidal method. These methods allow the inversion of a portion of the matrix, such as the diagonal, rather than no inversion at all. To invert the diagonal matrix requires a simple computational process.

A considerable amount of information is known about the second partial derivative matrix as eluded to in appendix D. For instance, the concavity of the matrix is solely dependent upon the weighted input matrix. Hence, the more we know about the input, the more we know about the concavity of the performance surface. Second order techniques use the concavity of the performance surface in computing a new update to the parameter being optimized. An example of where this discussion is heading is provided. Suppose that the input components were orthogonal to one another. This provides an orthogonal matrix. With this added information on the input, it would be a simple process to invert the matrix, since

$$A^{-1} = A^{T}.$$

2. The algorithm also approximated the average second partial derivative of the performance surface as it's instantaneous value. A better approach may be to calculate the average second partial matrix or some portion of it. A means of computing this average may take the following form:

$$\text{avg}\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) = \mu \cdot \sum_{k=0}^{\nu} \left. \frac{\partial^2 s}{\partial w \partial w^T} \right|_k \cdot \exp(-\mu \cdot (\nu - k)),$$

where ν the represents the maximum number of time steps desired to average, while k represents the current time step. The above equation provides a weighted memory of the past, with the most recent input receiving the most weight.

Another alternative would be to explore a batch technique. The idea is to process all of the input data and then average the total output error and corresponding partials. The respective update equation would then have information pertaining to all inputs and again provide somewhat of a memory of all input data from the environment.

3. A more thorough investigation of the Roggemann FLIR features is required. The number features used for target and non-target classification could be increased from the three used in this study. Later findings by Roggemann show that the Bayesian classifier improved dramatically by increasing the number features (to 9). The neural network performance should improve given the additional information on the input.

6.3. Conclusions

The second order approximation to Newton's method proved quite successful in pattern classification applications. In some instances, it slightly accelerated convergence. The network was able to classify targets with a moderately high degree of accuracy. The classification of features segmented from the FLIR imagery, is truly exciting. Each method provided classification accuracies of the test data at close to 85% and near perfect accuracy on the training data, when using the moment invariants as features.

There were four basic contributions made during this thesis effort. First of all, this research effort has provided, tested and validated a new biologically based neural network classifier. The network applies second derivative information concerning the second partial derivatives of the performance surface (in this case error surface). In addition to providing second derivative information this algorithm also provides information about how the surface is changing with time. Even though the algorithm did not provide a significant improvement in convergence time or accuracy, it still performed on a comparable level with the momentum method. This result alone adds validity to the concepts behind the algorithm and continued study in this area is warranted. Furthermore, this algorithm allowed for an easy comparison between three different minimization techniques. The results of this thesis clearly demonstrates the advantages of using the momentum and second order methods over the steepest descent method.

Secondly, the success of the artificial neural network classifier reinforces the fact that they can be very effective in applications on automated target recognition. In comparison with the Bayesian classifier, results demonstrate that the neural network classifier exceeded the performance of the statistical classifier on the training data. However, the Bayesian classifier performed better on the test data. When the classification criterion was less stringent (and comparable to the Bayesian criterion), the neural net classifier using the second order method further exceeded training performance levels. The test data accuracy approached the performance levels of the Bayesian classifier. This can be observed from the results in Appendix E. These results reinforce earlier results found by Ruck [12], demonstrating the superior performance of neural net classifiers over statistical classifiers.

The third contribution concerns network generalization. Results show that there may be a definite dividing line between the network actually learning and memorizing its environment over continuous training. Once that dividing line is crossed the network begins to memorize, thus destroying it's ability to generalize or learn from its environment. In this study, the test data accuracy began to deteriorate. This was particularly noticeable with the FLIR imagery features. When the classifier processed the doppler imagery features, this phenomenon was not as noticeable. This should not be terribly disturbing, since the data base consisting of doppler imagery features was so heavily influenced by tank features. The network may not have seen the other features enough to draw on such a conclusion.

The fourth and final contribution reveals that the

classifier has the same disadvantages as a human observer. For instance, when the field of view was wide, the resolution of the object of interest was poor. The actual target was indistinguishable by a human observer. Using a narrower field of view, provided better resolution and the target blobs were very distinct. When using features generated from wide and narrow fields of view the classification accuracy never grew higher than 63% on the training data. After removing the objects from the wide field of view, classification increased dramatically.
In a supervised training environment, feeding the network a poorly resolved object with a classification label, is the same as lying to the network.

Appendix A: An Introduction to Newton's Method and Iterative Methods

In chapter two, the approach to Parker's second order derivation [8] was introduced. In this appendix, the steps which were left out are provided here in detail. The first section will cover the missing steps in the derivation of the second order Newton's method. This will immediately be followed by an introductory discussion on Newton's method in general. Parker chose to approximate Newton's method, by solving the linear differential equation by applying an iterative approach. Therefore, an iterative approach will be discussed in general, in the third section. The final topic for discussion concerns convergence and the addition of Parker's leakage terms to the approximation derived from the iterative approach. The following text is the result of conversations and notes taken from interviews with Dr. Mark Oxley [6].

A.1. Derivation of Newton's Method

The derivation begins with a restatement of Eq. 2.4, where the functional dependencies of s on t, $f_{in}(\tau)$, and $w(t)$ have been suppressed for convenience. The average instantaneous performance is given by:

$$avg(s) = \mu \cdot \int_{-\infty}^{t} s \cdot e^{-\mu(t-\tau)} \, d\tau .$$

Assume that t is fixed and the above equation will provide a snapshot of the performance surface. The equation for the optimum weights is being derived from an optimum criterion. This implies, the derivative of the average performance surface with respect to the weights, evaluated at the optimum weights (w^*) is zero. Temporarily, let

$$q = \frac{\partial avg(s)}{\partial w^*} = 0 = \mu \cdot \int_{-\infty}^{t} \frac{\partial s}{\partial w^*} \cdot e^{-\mu(t-\tau)} \, d\tau. \qquad \text{A. 1}$$

Now, let t vary and q becomes a constant function of time. Again, from an optimum perspective, as s changes the weights continue to follow the moving minimum. The next step is to apply Leibniz's rule and compute the time derivative of Eq. A.1, where

$$\frac{\partial q}{\partial t} = 0 = \mu \cdot \frac{\partial s}{\partial w^*} + \mu \cdot \int_{-\infty}^{t} \frac{\partial}{\partial t}\left(\frac{\partial s}{\partial w^*} \right) \cdot e^{-\mu(t-\tau)} \, d\tau$$

$$- \mu^2 \cdot \int_{-\infty}^{t} \frac{\partial s}{\partial w^*} \cdot e^{-\mu(t-\tau)} \, d\tau.$$

Notice that the second integral term is equal to $-\mu q$, and therefore equal to zero. In the first integral, the only way s depends on t is through **w**(t) and not on the input which depends on τ. Therefore, the following relationship holds:

$$\frac{\partial}{\partial t}\left(\frac{\partial s}{\partial w^*} \right) = \left(\frac{\partial^2 s}{\partial w^* \partial w^{*T}} \right) \cdot \frac{\partial w^*}{\partial t}$$

such that $\partial q / \partial t$ becomes·

$$0 = \mu\cdot\frac{\partial s}{\partial w^*} + \left(\mu\cdot\int_{-\infty}^{t}\frac{\partial^2 s}{\partial w^*\partial w^{*T}}\cdot e^{-\mu(t-\tau)}\,d\tau\right)\cdot\frac{\partial w^*}{\partial t}$$

and

$$0 = \mu\cdot\frac{\partial s}{\partial w^*} + \text{avg}\left(\frac{\partial^2 s}{\partial w^*\partial w^{*T}}\right)\cdot\frac{\partial w^*}{\partial t}.$$

The exponentially weighted time average of the second derivative is a matrix (see appendix B). If the above matrix is invertible (see appendix D) an explicit first order differential equation for the optimum path of the weights is:

$$\frac{\partial w^*}{\partial t} = -\left[\text{avg}\left(\frac{\partial^2 s}{\partial w^*\partial w^{*T}}\right)\right]^{-1}\cdot\mu\cdot\frac{\partial s}{\partial w^*} \qquad \text{A. 2}$$

Thus Eq. 2.5 has been derived. Equation A.2 is a second order Newton's method.

A.2. Newton's Method in General

Newton's method is an iteration method for solving equations of the form f(x) = 0, where f(x) has a continuous derivative. The method is commonly used because of its simplicity and great speed to convergence. The general idea is to approximate the graph of f(x) by suitable tangents, see Fig. A.1.

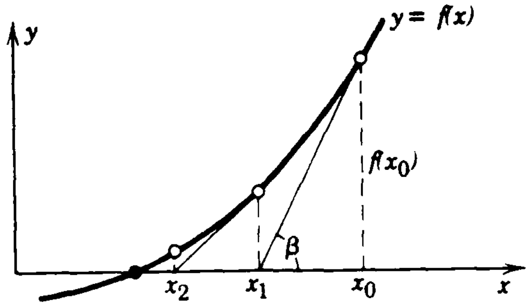


Figure A.1. An illustrative graph of Newton's method [3]

Using an approximate value x_0 obtained from the graph of f(x), let x_1 be the point of intersection of the x-axis and the tangent to the curve at $f(x_0)$. Then

$$\tan \beta = \frac{df(x)}{dx} = \frac{f(x_0)}{x_0 - x_1}$$

where

$$x_1 = x_0 - \frac{f(x_0)}{\frac{df(x_0)}{dx}}.$$

In the second step, x_2 is found from x_1 and in the following step x_3 is found from x_2. This is performed until $f(x_K) = 0$. In general, Newton's method becomes:

$$x_{K+1} = x_K - \frac{f(x_K)}{\dfrac{df(x_K)}{dx}}.$$

Equation A.2 is a straight forward continuum extension of the above Newton's method. It's desired to have the partial of s with respect to $\mathbf{w}^*$ approach 0.

A.3. An Iterative Method for Ax = b

If the matrix of Eq. A.2 is not invertible (and it is not) or calculating the second order Newton's method is tedious, there is another approach.

Consider the following expression given by

$$\mathbf{A}(t, \mathbf{x}(t))\cdot\mathbf{x}(t) = \mathbf{b},$$

which is in the same form as Eq A.2, with the matrix on the right hand side of the equation. Let $\mathbf{Ax} = \mathbf{b}$, to reduce the notational overhead. One approach to approximating the vector $\mathbf{x}$ is to perform an iterative method. First, multiply the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ by $\beta\Delta t$. The β controls the rate of convergence and Δt is a small time increment which will later go to zero. Then add and subtract the identity matrix from the resulting $\mathbf{A}$ matrix. This is a means of splitting the $\mathbf{A}$ matrix and is accomplished in the following manner:

$$(\ \beta\cdot\Delta t\cdot\mathbf{A} + \mathbf{I} - \mathbf{I}\ )\cdot\mathbf{x} = \beta\cdot\Delta t\cdot\mathbf{b},$$

and

$$\mathbf{x} : \mathbf{x} - \beta\cdot\Delta t\cdot\mathbf{A}\cdot\mathbf{x} + \beta\cdot\Delta t\cdot\mathbf{b}.$$

The next step allows the iterative approach to take full form. The idea is to use the partial time derivative of **x**(t) to compute an improved estimate of the partial time derivative of **x**(t+Δt). It is desired to use some known vector to predict an improved version of that same vector. It is hoped, by performing this process in an iterative manner, the time derivative of **x**(t+Δt) will eventually be obtained. Returning the functional dependencies for clarity, the iterative approach assumes the following form:

$$\mathbf{x}(t+\Delta t) : \mathbf{x}(t) - \beta\cdot\Delta t\cdot\mathbf{A}(t,\ \mathbf{x}(t))\cdot\mathbf{x}(t) + \beta\cdot\Delta t\cdot\mathbf{b},$$

where

$$\frac{\mathbf{x}(t+\Delta t) - \mathbf{x}(t)}{\Delta t} : - \beta\cdot\mathbf{A}(t,\ \mathbf{x}(t))\cdot\mathbf{x}(t) + \beta\cdot\mathbf{b}$$

In the limit as Δt approaches zero, defines the partial time derivative of **x**(t), that is

$$\frac{\partial\mathbf{x}(t)}{\partial t} : - \beta\cdot\mathbf{A}(t,\ \mathbf{x}(t))\cdot\mathbf{x}(t) + \beta\cdot\mathbf{b}.$$

Let

$$\mathbf{x}(t) : \frac{\partial\mathbf{w}^{+}}{\partial t},\quad \mathbf{A}(t,\ \mathbf{x}(t)) : \mathrm{avg}\left(\frac{\partial^2 s}{\partial\mathbf{w}^{+}\partial\mathbf{w}^{+T}}\right),\quad \mathbf{b} : -\mu\cdot\frac{\partial s}{\partial\mathbf{w}^{+}}$$

and then

$$\frac{\partial^2 \mathbf{w}^+(t)}{\partial t^2} = -\beta \cdot \text{avg}\left(\frac{\partial^2 s}{\partial \mathbf{w}^+ \partial \mathbf{w}^{+T}}\right) \cdot \frac{\partial \mathbf{w}^+(t)}{\partial t} - \beta \cdot \mu \cdot \frac{\partial s}{\partial \mathbf{w}^+}. \qquad \text{A. 3}$$

The (+) notation is used to imply an approximate value. Hence, Eq. 2.7 is derived.

A.4. Addition of Leakage Terms and Convergence

From Eq. A.3, Parker adds leakage terms to insure the network will converge to some minimum, since the iterative approach is no more than an approximation [7:593-600; 8]. Consider the following argument. Suppose that the iterative approach succeeds in driving $\mathbf{x}(t+\Delta t) - \mathbf{x}(t)$ to 0. Thus,

$$\mathbf{A} \cdot \mathbf{x}^+ = \mathbf{b}$$

implying that approximate $\mathbf{x}^+$ lies in a family of solutions consisting of linear combinations, if $\mathbf{A}$ is not invertible. In other words, there are a number of solutions for $\mathbf{x}^+$, which satisfy the above equation. Many of these solutions may lie in local minima. Thus, convergence to the optimum $\mathbf{x}^*$ may never be achieved. Therefore, a method must be sought to insure convergence.

Parker argues that a natural way to insure convergence is the addition of leakage terms [8]. It's assumed that the integrators required to implement the algorithm are indeed leaky. An analogy drawn by Parker concerns that of an analog circuit.

Consider electrons leaking off of a capacitor which stores energy in the form of charge. Since all practical integrators have a leakage rate, it seems logical to take them into consideration [8].

Consider Eq. A.3. The first step is to calculate, the second partial time derivative of $\mathbf{w}^+$, and then integrate to obtain the first partial time derivative of $\mathbf{w}^+$, denoted as $\mathbf{q}^+$. By integrating $\mathbf{q}^+$, $\mathbf{w}^+$ is obtained:

$$\mathbf{q}^+ = \int_{-\infty}^{t} \left(- \beta \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial \mathbf{w}^+ \partial \mathbf{w}^{+T}} \right) \cdot \mathbf{q}^+ - \beta \cdot \mu \cdot \frac{\partial s}{\partial \mathbf{w}^+} \right) d\tau,$$

$$\mathbf{w}^+ = \int_{-\infty}^{t} \mathbf{q}^+ \, d\tau.$$

Next, take the time derivative of each of the equations above:

$$\frac{\partial \mathbf{q}^+}{\partial t} = - \beta \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial \mathbf{w}^+ \partial \mathbf{w}^{+T}} \right) \cdot \mathbf{q}^+ - \beta \cdot \mu \cdot \frac{\partial s}{\partial \mathbf{w}^+},$$

$$\frac{\partial \mathbf{w}^+}{\partial t} = \mathbf{q}^+.$$

Since two integrations are performed, then two leakage terms are required. Therefore, subtracting the respective leakage terms from each of the above equations has the following result:

$$\frac{\partial \mathbf{q}}{\partial t} = - \beta \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial \mathbf{w} \partial \mathbf{w}^{T}} \right) \cdot \mathbf{q} - \beta \cdot \mu \cdot \frac{\partial s}{\partial \mathbf{w}} - l_2 \cdot \mathbf{q}, \qquad \text{A. 4}$$

$$\frac{\partial \mathbf{w}}{\partial t} = \mathbf{q} - l_1 \cdot \mathbf{w}. \qquad \text{A. 5}$$

The (+) notation has been removed, since it is now believed that a better approximation of the quantities of the left hand side of each equation has been found. It is now desired to remove the dependency on **q**, by taking the time derivative of Eq. A.5:

$$\frac{\partial^2 \mathbf{w}}{\partial t^2} = \frac{\partial \mathbf{q}}{\partial t} - l_1 \cdot \frac{\partial \mathbf{w}}{\partial t} \qquad \text{A. 6}$$

and substituting Eq. A. 2 into Eq. A. 4:

$$\frac{\partial^2 \mathbf{w}}{\partial t^2} = -\beta \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial \mathbf{w} \partial \mathbf{w}^T} \right) \cdot \mathbf{q} - \beta \cdot \mu \cdot \frac{\partial s}{\partial \mathbf{w}} - l_2 \cdot \mathbf{q} - l_1 \cdot \frac{\partial \mathbf{w}}{\partial t} \qquad \text{A. 7}$$

and finally, rewriting Eq. A. 5, such that

$$\mathbf{q} = \frac{\partial \mathbf{w}}{\partial t} + l_1 \cdot \mathbf{w}$$

and substitute **q** into Eq. A. 5, which yields the final result:

$$\frac{\partial^2 \mathbf{w}}{\partial t} = -\beta \cdot \mu \cdot \frac{\partial s}{\partial \mathbf{w}} - \left(l_1 \cdot l_2 \cdot \mathbf{I} + \beta \cdot l_1 \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial \mathbf{w} \partial \mathbf{w}^T} \right) \right) \cdot \mathbf{w}$$

$$- \left((l_1 + l_2) \cdot \mathbf{I} + \beta \cdot \mathrm{avg}\left(\frac{\partial^2 s}{\partial \mathbf{w} \partial \mathbf{w}^T} \right) \right) \cdot \frac{\partial \mathbf{w}}{\partial t}. \qquad \text{A. 8}$$

To simplify the above equation somewhat, let

$$a_1 = \beta \cdot \mu,$$

$$a_2 : l_1 \cdot l_2,$$

$$a_3 : B \cdot l_1,$$

$$a_4 : l_1 + l_2, \text{ and}$$

$$a_5 : \beta.$$

Reparameterizing Eq. A.6, results in the following:

$$\frac{\partial^2 w}{\partial t} : -a_1 \cdot \frac{\partial s}{\partial w} - \left(a_2 \cdot I + a_3 \cdot avg\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \right) \cdot w - \left(a_4 \cdot I + a_5 \cdot avg\left(\frac{\partial^2 s}{\partial w \partial w^T} \right) \right) \cdot \frac{\partial w}{\partial t}. \qquad \text{A. 9}$$

The result of adding the leakage terms, is that it provides the *same* effect as adding a momentum term (see appendix D) and additive noise. The momentum term has the effect of smoothing the error surface. The basic concept behind momentum, is to suppress local minima, and enhance the global minimum. The partial time derivative of **w** associated with a_4 is used in introducing the momentum term (see section 3.4.2). The terms associated with a_2 and a_3, combine to introduce noise into the network.

Appendix B: Linear Algebraic Forms and Notation

In chapters two and three, the equations introduced were heavily dependent on linear algebraic forms. This appendix serves as an attempt to clear up the notational overhead. This section also provides examples of elementary linear algebra in the form of matrix addition and multiplication. In the discussions below, each vector is considered a column vector unless otherwise specified as the vector transpose.

The problem posed, is to expand the first and second order partial derivatives of the performance quantity (s) and expose the impending linear algebra. In chapter three, the network performance quantity (s) was introduced as the squared error. Consider the first partial derivative of the network performance indicator (s). The partial derivative of s, as introduced in chapter three, has the following form when written in terms of matrices,

$$\frac{\partial s}{\partial \mathbf{w}} = 2 \cdot \frac{\partial \mathbf{e}^T}{\partial \mathbf{w}} \cdot \mathbf{e}$$

$$= 2 \cdot \begin{bmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_2}{\partial w_1} & \cdots & \frac{\partial e_m}{\partial w_1} \\ \frac{\partial e_1}{\partial w_2} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_m}{\partial w_2} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_1}{\partial w_n} & \frac{\partial e_2}{\partial w_n} & \cdots & \frac{\partial e_m}{\partial w_n} \end{bmatrix} \cdot \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}$$

where

$$\begin{bmatrix} \frac{\partial s}{\partial w_1} \\ \frac{\partial s}{\partial w_2} \\ \vdots \\ \frac{\partial s}{\partial w_n} \end{bmatrix} = 2 \cdot \begin{bmatrix} \Sigma_l \frac{\partial e_l}{\partial w_1} \cdot e_l \\ \Sigma_l \frac{\partial e_l}{\partial w_2} \cdot e_l \\ \vdots \\ \Sigma \frac{\partial e_l}{\partial w_n} \cdot e_l \end{bmatrix}. \qquad \text{B. 1}$$

The partial derivative of s with respect to **w** is a column vector. When the performance indicator is defined as error, the first partial is expressed as the sum of the partials of each error signal with respect to a given weight. The partial with respect to each weight in the network, is the direction of the gradient and points toward the maximum of the performance surface.

Therefore, the discussion begins with the second partial of s as defined in chapter three,

$$\frac{\partial^2 s}{\partial w \partial w^T} = 2\cdot\left(\frac{\partial e^T}{\partial w}\cdot\frac{\partial e}{\partial w^T} + \frac{\partial^2 e^T}{\partial w \partial w^T}\cdot e\right). \qquad \text{B. 2}$$

The above second partial is a matrix as is each of its components. Below each component matrix is defined and expanded. First, consider the partial of **e** transpose with respect to the weight vector, where

$$\frac{\partial e^T}{\partial w} = \begin{bmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_2}{\partial w_1} & \cdots & \frac{\partial e_m}{\partial w_1} \\ \frac{\partial e_1}{\partial w_2} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_m}{\partial w_2} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_1}{\partial w_n} & \frac{\partial e_2}{\partial w_n} & \cdots & \frac{\partial e_m}{\partial w_n} \end{bmatrix} \qquad \text{B. 3}$$

and similarly, the partial of **e** with respect to **w** transpose

$$\frac{\partial e}{\partial w^T} = \begin{bmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \frac{\partial e_1}{\partial w_n} \\ \frac{\partial e_2}{\partial w_1} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_2}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_m}{\partial w_1} & \frac{\partial e_m}{\partial w_2} & \cdots & \frac{\partial e_m}{\partial w_n} \end{bmatrix}. \qquad \text{B. 4}$$

The product of the matrices, Eqs. B.3 and B.4, determines the first component of the sum in Eq. B.2.

$$\frac{\partial e^T}{\partial w}\cdot\frac{\partial e}{\partial w^T} = \begin{bmatrix} \sum_l \frac{\partial e_l}{\partial w_1}\cdot\frac{\partial e_l}{\partial w_1} & \sum_l \frac{\partial e_l}{\partial w_1}\cdot\frac{\partial e_l}{\partial w_2} & \cdots & \sum_l \frac{\partial e_l}{\partial w_1}\cdot\frac{\partial e_l}{\partial w_n} \\ \sum_l \frac{\partial e_l}{\partial w_2}\cdot\frac{\partial e_l}{\partial w_1} & \sum_l \frac{\partial e_l}{\partial w_2}\cdot\frac{\partial e_l}{\partial w_2} & \cdots & \sum_l \frac{\partial e_l}{\partial w_2}\cdot\frac{\partial e_l}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \sum_l \frac{\partial e_l}{\partial w_n}\cdot\frac{\partial e_l}{\partial w_1} & \sum_l \frac{\partial e_l}{\partial w_n}\cdot\frac{\partial e_l}{\partial w_2} & \cdots & \sum_l \frac{\partial e_l}{\partial w_n}\cdot\frac{\partial e_l}{\partial w_n} \end{bmatrix}. \qquad \text{B. 5}$$

The next component considered is the second partial of e transpose with respect to w and w transpose. This matrix is a little more complicated and care must be taken to insure indices are maintained when multiplying by e. This last component of the second partial of (s) is defined as

$$\frac{\partial^2 e^T}{\partial w \partial w^T} \cdot e = \sum_{l=1}^{m} e_l \cdot \frac{\partial^2 e_l}{\partial w_i \partial w_j}$$

$$= \begin{bmatrix} \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_1 \partial w_1} & \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_1 \partial w_2} & \cdots & \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_1 \partial w_n} \\ \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_2 \partial w_1} & \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_2 \partial w_2} & \cdots & \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_2 \partial w_n} \\ \vdots & \vdots & & \vdots \\ \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_n \partial w_1} & \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_n \partial w_2} & \cdots & \Sigma_l \; e_l \cdot \frac{\partial^2 e_l}{\partial w_n \partial w_n} \end{bmatrix}. \tag{B. 6}$$

Appendix C: Partial Derivatives of the Sigmoid Function

This appendix contains all the significant partial derivatives of the sigmoid function used in this study of artificial neural networks. Recall, that the output of a single cell was chosen to be the sigmoid function, as introduced in chapter two. In the various implementation stages it was necessary to compute the first and second partial derivatives with respect various independent variables. The independent variables considered were the inputs to the cell (f_{in}), the interconnection weights (w), and the time (t) variables.

For clarity and simplicity, a single cell is considered with several inputs and of course a single output, as shown in Fig. C.1.

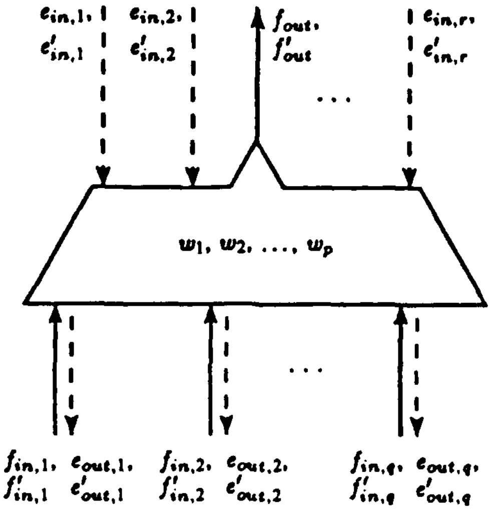


Figure C.1 A Single Cell

The first order partials will be considered first, followed by the second order partials. The computations begin with the partial of the output (f_{out}) with respect to **w**.

$$\frac{\partial f_{out}}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}\left(\frac{1}{1 + \exp(-\mathbf{f}_{in}^{T} \cdot \mathbf{w})} \right)$$

$$= \frac{\partial}{\partial \mathbf{w}}\left(\left(1 + \exp(-\mathbf{f}_{in}^{T} \cdot \mathbf{w}) \right)^{-1} \right)$$

$$= -1 \cdot \left(1 + \exp(-\mathbf{f}_{in}^{T} \cdot \mathbf{w}) \right)^{-2} \cdot \frac{\partial}{\partial \mathbf{w}}\left(\exp(-\mathbf{f}_{in}^{T} \cdot \mathbf{w}) \right)$$

$$= -\frac{1}{1 + \exp(-\mathbf{f}_{in}^{T} \cdot \mathbf{w})} \cdot \frac{\exp(-\mathbf{f}_{in}^{T} \cdot \mathbf{w})}{1 + \exp(-\mathbf{f}_{in}^{T} \cdot \mathbf{w})} \cdot \frac{\partial}{\partial \mathbf{w}}(\mathbf{f}_{in}^{T} \cdot \mathbf{w})$$

Using the following relationship, where

$$\frac{\exp(-\alpha)}{1 + \exp(-\alpha)} = 1 - \frac{1}{1 + \exp(-\alpha)}$$

$$= 1 - f_{out}$$

and

$$\alpha = \mathbf{f}_{in}^{T} \cdot \mathbf{w}.$$

Taking the above substitution, the partial of f_{out} with respect to **w** becomes:

$$\frac{\partial f_{out}}{\partial \mathbf{w}} = f_{out} \cdot \left(1 - f_{out} \right) \cdot \mathbf{f}_{in}. \qquad \text{C. 1}$$

For example, the partial of f_{out} with respect to w_1 is

$$\frac{\partial f_{out}}{\partial w_1} = f_{out} \cdot \left(1 - f_{out} \right) \cdot f_{in,1} .$$

Similarly, the partial of f_{out} with respect to f_{in} has the same form and is expressed as:

$$\frac{\partial f_{out}}{\partial f_{in}} = f_{out} \cdot \left(1 - f_{out} \right) \cdot w \qquad \text{C. 2}$$

As for the time derivative of the sigmoid function, it is assumed that both the inputs and weights are functions of time. Therefore, the derivative of the sigmoid with respect to time will again be a partial over all the weights and inputs of the cell, such that

$$\frac{\partial f_{out}}{\partial t} = \frac{\partial}{\partial t}\left(\frac{1}{1 + \exp(-f^{T}_{in} \cdot w)} \right)$$

$$= \frac{\partial}{\partial t}\left(\left(1 + \exp(-f^{T}_{in} \cdot w) \right)^{-1} \right)$$

$$= -1 \cdot \left(1 + \exp(-f^{T}_{in} \cdot w) \right)^{-2} \cdot \frac{\partial}{\partial t}\left(\exp(-f^{T}_{in} \cdot w) \right)$$

$$= - \frac{1}{1 + \exp(-f^{T}_{in} \cdot w)} \cdot \frac{\exp(-f^{T}_{in} \cdot w)}{1 + \exp(-f^{T}_{in} \cdot w)} \cdot \frac{\partial}{\partial t}(-f^{T}_{in} \cdot w)$$

$$= f_{out} \cdot \left(1 - f_{out} \right) \cdot \frac{\partial}{\partial t}\left(f^{T}_{in} \cdot w \right) .$$

From here the chain rule is applied to obtain the following:

$$\frac{\partial f_{out}}{\partial t} = f_{out} \cdot (1 - f_{out}) \cdot \left(f^{T}_{in} \cdot \frac{\partial w}{\partial t} + \frac{\partial f^{T}_{in}}{\partial t} \cdot w \right).$$

The above may be rewritten by applying Eqs. C.1 and C.2, where

$$\frac{\partial f_{out}}{\partial t} = \frac{\partial f_{out}}{\partial w^{T}} \cdot \frac{\partial w}{\partial t} + \frac{\partial f_{out}}{\partial f^{T}_{in}} \cdot \frac{\partial f_{in}}{\partial t}. \qquad \text{C. 3}$$

The second order partials are a straight forward extension of the first order partials. In fact, Eqs. C.1 and C.2 will be used in the computation of the second order partials. To begin, the partial of f_{out} with respect to w and w transpose is considered.

$$\frac{\partial^2 f_{out}}{\partial w \partial w^{T}} = \frac{\partial}{\partial w} \left(\frac{\partial f_{out}}{\partial w^{T}} \right)$$

$$= \frac{\partial}{\partial w} \left(f_{out} \cdot (1 - f_{out}) \cdot f^{T}_{in} \right)$$

$$= \left(\frac{\partial f_{out}}{\partial w} \cdot (1 - f_{out}) - f_{out} \cdot \frac{\partial f_{out}}{\partial w} \right) \cdot f^{T}_{in}$$

$$= \left(f_{out} \cdot (1 - f_{out})^{2} \cdot f_{in} - (f_{out})^{2} \cdot (1 - f_{out}) \cdot f_{in} \right) \cdot f^{T}_{in}$$

$$= f_{out} \cdot (1 - f_{out}) \cdot \Big((1 - f_{out}) - f_{out} \Big) \cdot f_{in} \cdot f^{T}_{in}$$

$$\frac{\partial^2 f_{out}}{\partial w \partial w^T} = f_{out} \cdot (1 - f_{out}) \cdot (1 - 2 \cdot f_{out}) \cdot f_{in} \cdot f_{in}^T \qquad \text{C. 4}$$

Similarly, the second partial of f_{out} with respect to f_{in} and f_{in} transpose is found to be:

$$\frac{\partial^2 f_{out}}{\partial f_{in} \partial f_{in}^T} = f_{out} \cdot (1 - f_{out}) \cdot (1 - 2 \cdot f_{out}) \cdot w \cdot w^T. \qquad \text{C. 5}$$

Another second partial used in the formulation of the second order algorithm, is the second partial of f_{out} with respect to w and f_{in} transpose. Again the use of Eqs. C.1 and C.2, and the application of the chain rule is desired, to obtain

$$\frac{\partial f_{out}}{\partial w \partial f_{in}^T} = \frac{\partial}{\partial w}\left(\frac{\partial f_{out}}{\partial f_{in}^T} \right)$$

$$= \frac{\partial}{\partial w}\left(f_{out} \cdot (1 - f_{out}) \cdot w^T \right)$$

$$= \left(\frac{\partial f_{out}}{\partial w} \cdot (1 - f_{out}) - f_{out} \cdot \frac{\partial f_{out}}{\partial w} \right) \cdot w^T$$

$$+ f_{out} \cdot (1 - f_{out}) \cdot \frac{\partial}{\partial w}\left(w^T \right)$$

where the result of the partial of w^T with respect to w is an (nxn) matrix and is denoted as I, and has the following form:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & 1 \end{bmatrix}.$$

With this in mind the final form of the second partial becomes:

$$\frac{\partial^2 f_{out}}{\partial \mathbf{w} \partial \mathbf{f}_{in}^T} = \left(f_{out} \cdot (1 - f_{out})^2 \cdot \mathbf{f}_{in} - (f_{out})^2 \cdot (1 - f_{out}) \cdot \mathbf{f}_{in} \right) \cdot \mathbf{w}^T$$

$$+ f_{out} \cdot (1 - f_{out}) \cdot \mathbf{I}$$

$$= f_{out} \cdot (1 - f_{out}) \cdot \left((1 - f_{out}) - f_{out} \right) \cdot \mathbf{f}_{in} \cdot \mathbf{w}^T$$

$$+ f_{out} \cdot (1 - f_{out}) \cdot \mathbf{I}$$

$$\frac{\partial^2 f_{out}}{\partial \mathbf{w} \partial \mathbf{f}_{in}^T} = f_{out} \cdot (1 - f_{out}) \cdot (1 - 2 \cdot f_{out}) \cdot \mathbf{f}_{in} \cdot \mathbf{w}^T$$

$$+ f_{out} \cdot (1 - f_{out}) \cdot \mathbf{I} \qquad \text{C. 6}$$

In a similar fashion, the second partial of f_{out} with respect to $\mathbf{f}_{in}$ and $\mathbf{w}^T$ is found to be

$$\frac{\partial^2 f_{out}}{\partial \mathbf{f}_{in} \partial \mathbf{w}^T} = f_{out} \cdot (1 - f_{out}) \cdot (1 - 2 \cdot f_{out}) \cdot \mathbf{w} \cdot \mathbf{f}_{in}^T$$

$$+ f_{out} \cdot (1 - f_{out}) \cdot \mathbf{I} \qquad \text{C. 7}$$

The final partials to consider are the partials of f_{out} with respect to time and **w** or f_{in}.

$$\frac{\partial}{\partial t}\left(\frac{\partial f_{out}}{\partial \mathbf{w}}\right) = \frac{\partial}{\partial \mathbf{w}}\left(\frac{\partial f_{out}}{\partial t}\right)$$

$$= \frac{\partial}{\partial \mathbf{w}}\left(\frac{\partial f_{out}}{\partial \mathbf{w}^T}\cdot\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial f_{out}}{\partial \mathbf{f}^T_{in}}\cdot\frac{\partial \mathbf{f}_{in}}{\partial t}\right),$$

by applying Eq. C. 3.

$$\frac{\partial}{\partial \mathbf{w}}\left(\frac{\partial f_{out}}{\partial t}\right) = \frac{\partial^2 f_{out}}{\partial \mathbf{w}\partial \mathbf{w}^T}\cdot\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial^2 f_{out}}{\partial \mathbf{w}\partial \mathbf{f}^T_{in}}\cdot\frac{\partial f_{in}}{\partial t}$$

$$= f_{out}\cdot(1 - f_{out})\cdot(1 - 2\cdot f_{out})\cdot \mathbf{f}_{in}\cdot \mathbf{f}^T_{in}\cdot\frac{\partial \mathbf{w}}{\partial t}$$

$$+ f_{out}\cdot(1 - f_{out})\cdot(1 - 2\cdot f_{out})\cdot \mathbf{f}_{in}\cdot \mathbf{w}^T\cdot\frac{\partial \mathbf{f}_{in}}{\partial t}$$

$$+ f_{out}\cdot(1 - f_{out})\cdot\frac{\partial \mathbf{f}_{in}}{\partial t}$$

Similarly,

$$\frac{\partial}{\partial f_{in}}\left(\frac{\partial f_{out}}{\partial t}\right) = \frac{\partial^2 f_{out}}{\partial f_{in}\,\partial w^T}\cdot\frac{\partial w}{\partial t} + \frac{\partial^2 f_{out}}{\partial f_{in}\,\partial f_{in}^T}\cdot\frac{\partial f_{in}}{\partial t}$$

$$= f_{out}\cdot(\ 1 - f_{out}\ )\cdot(\ 1 - 2\cdot f_{out}\ )\cdot w\cdot f_{in}^T\cdot\frac{\partial w}{\partial t}$$

$$+ f_{out}\cdot(\ 1 - f_{out}\ )\cdot(\ 1 - 2\cdot f_{out}\ )\cdot w\cdot w^T\cdot\frac{\partial f_{in}}{\partial t}$$

$$+ f_{out}\cdot(\ 1 - f_{out}\ )\cdot\frac{\partial f_{in}}{\partial t}. \qquad \text{C. 8}$$

The above computations provide an excellent review, as well as a quick reference to the partial derivatives of the sigmoid function. From the results, it is readily seen why the sigmoid function is a popular nonlinear transfer function used by the artificial neural network community. All of the partials of the sigmoid are functions of the sigmoid itself. This makes computations very simple and convenient for a digital computer.

Appendix D: Second Order Convergence Conditions for a Single Cell

The topic of this appendix concerns some ideas for improving the network convergence to an optimum set of weights. More specifically, the questions posed are: Can the network begin its training routine with the second order algorithm? If so, what criterion must be met to insure that the network will converge on an optimum set of weights? Is there a means of improving the convergence times? The first section discusses the initial state of the network. It introduces the criterion which must be met in order to begin training with the second order algorithm. The next and final section entertains the idea of accelerating convergence with a momentum term. The following text is the result of conversations and notes taken from interviews with Dr. Mark Oxley [5].

D.1. Initialize Training with the Second Order Algorithm

To begin answering the above questions, a simple problem is constructed for clarity. Consider a single layer perceptron that classifies an analog input vector into two classes denoted 1 and 2, see Fig. D.1.

The single cell is to divide the space spanned by the input into two regions separated by a line (or hyperplane) in two dimensions. Class 1 will be represented by a desired output of 1, while the desired output for class 2 is 0. For training purposes it is desired to minimize the squared error function

with the second order algorithm. To begin training with a second

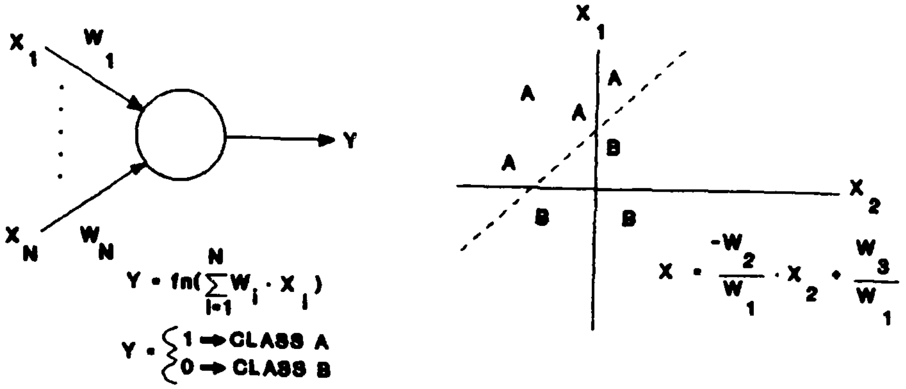

X_1
W_1
Y
X_N
W_N
$Y = fn(\sum_{i=1}^{N} W_i \cdot X_i)$
$Y = \begin{cases} 1 \rightarrow \text{CLASS A} \\ 0 \rightarrow \text{CLASS B} \end{cases}$
X_1
A
A
A
A
B
B
B
X_2
$X = \frac{-W_2}{W_1} \cdot X_2 + \frac{W_3}{W_1}$

Figure D.1 Pictorial Problem Description

order algorithm, it is desired for the initial weight settings to exist within the basin of attraction of a global minimum of the squared error performance surface. In this context, the global minimum is defined over the entire training ensemble of input vectors. If this criterion is not met, the path towards the optimum set of weights may never be found by the second order algorithm.

Let

$$s(w) = \frac{1}{k} \sum_{j=1}^{k} \left(d - f(x^j, w) \right)^2 \qquad \text{D. 1}$$

where s(w) is the average squared error over all k input vectors. For the problem considered here, **w** = (w_1, w_2, w_3) and **x** = (x_1, x_2, 1). Keep in mind that the concept can be extended to a network of higher dimensionality. By taking the first partial derivative of s and evaluating at the optimum weights ($\mathbf{w}^*$), it is desired that a minimum exists, and preferably equal to 0. Assume that $f(\mathbf{x}^j, \mathbf{w})$ is the sigmoid function, so that the results of appendix C may be used and

$$\frac{\partial s(\mathbf{w})}{\partial \mathbf{w}} = \frac{2}{k} \cdot \sum_{j=1}^{k} \left(d - f(\mathbf{x}^j, \mathbf{w}) \right) \cdot \frac{\partial}{\partial \mathbf{w}} \left(d - f(\mathbf{x}^j, \mathbf{w}) \right)$$

$$= - \frac{2}{k} \cdot \sum_{j=1}^{k} \left(d - f(\mathbf{x}^j, \mathbf{w}) \right) \cdot f(\mathbf{x}^j, \mathbf{w}) \cdot \left(1 - f(\mathbf{x}^j, \mathbf{w}) \right) \cdot \mathbf{x}^j \qquad \text{D. 2}$$

where it is desired for

$$\frac{\partial s(\mathbf{w}^*)}{\partial \mathbf{w}} = 0,$$

since the global minimum is also desired. To insure a minimum and not a saddle point, the second partial of s is considered.

$$\frac{\partial^2 s(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^T} = \frac{2}{k} \cdot \sum_{j=1}^{k} \frac{\partial}{\partial \mathbf{w}} \left((d - f(\mathbf{x}^j, \mathbf{w}) \cdot \frac{\partial}{\partial \mathbf{w}^T} \left(d - f(\mathbf{x}^j, \mathbf{w}) \right) \right)$$

$$= \frac{2}{k} \cdot \sum_{j=1}^{k} f(\mathbf{x}^j, \mathbf{w}) \cdot \left(1 - f(\mathbf{x}^j, \mathbf{w}) \right) \cdot \left\{ f(\mathbf{x}^j, \mathbf{w}) \cdot \left(1 - f(\mathbf{x}^j, \mathbf{w}) \right) - \left(d - f(\mathbf{x}^j, \mathbf{w}) \right) \cdot \left(1 - 2 \cdot f(\mathbf{x}^j, \mathbf{w}) \right) \right\} \cdot \mathbf{x}^j \cdot (\mathbf{x}^j)^T. \qquad \text{D. 3}$$

The expression preceeding the matrix is a scalar for given values of $\mathbf{x}^j$ and $\mathbf{w}$; therefore, the entire expression is in the form of a matrix. If the expression is determined to be positive definite, then when the first partial of s is evaluated at the optimum weights the result is the minimum of the error surface, and ideally zero.

First, it must be determined that $\mathbf{x}^J \cdot (\mathbf{x}^J)^T$ is a positive definite matrix or not. Therefore, a necessary and sufficient condition for the real symmetric matrix **A** to be positive definite [14:243-254]:

(1) $\mathbf{y}^T \cdot \mathbf{A} \cdot \mathbf{y} > 0$ for all nonzero vectors **y**.

(2) All the eigenvalues of **A** satisfy $\lambda_1 > 0$.

(3) All the submatrices $\mathbf{A}_n$ have postive determinants.

(4) All the pivots (without row exchanges) satisfy $d_1 > 0$.

To satisfy criterion (1) consider the following:

$$\begin{aligned} \mathbf{y}^T \cdot \mathbf{A} \cdot \mathbf{y} &= \mathbf{y}^T \cdot \mathbf{x}^J \cdot (\mathbf{x}^J)^T \cdot \mathbf{y} \\ &= (\mathbf{y}^T \cdot \mathbf{x}^J) \cdot ((\mathbf{x}^J)^T \cdot \mathbf{y}) \\ &= (\mathbf{y}^T \cdot \mathbf{x}^J)^2 \geq 0. \end{aligned}$$

The above result shows that the matrix $\mathbf{x}^J \cdot (\mathbf{x}^J)^T$ could be positive definite or positve semi-definite (implying a 0 eigenvalue). Further investigation with criterion (2) is necessary. If the matrix is singular, then a 0 eigenvalue exists. To determine the singularity of a matrix consider the determinant of **A**.

Recall that $\mathbf{x}^J$ represents an arbitrary input pattern, where each pattern will be considered a column vector. Each component of a single pattern or vector is denoted as x_i. Considering a specific input vector to ease the notational overhead, and expanding the matrix yields:

$$\mathbf{x}\cdot\mathbf{x}^T = \begin{bmatrix} x_1\cdot x_1 & x_1\cdot x_2 & x_1 \\ x_2\cdot x_1 & x_2\cdot x_2 & x_2 \\ x_1 & x_2 & 1 \end{bmatrix}$$

and hence the

$$\begin{aligned} |\mathbf{A}| &= |\mathbf{x}\cdot\mathbf{x}^T| \\ &= (x_1)^2\cdot(\ (x_2)^2 - (x_2)^2\ ) - x_1\cdot x_2\cdot(\ x_2\cdot x_1 - x_2\cdot x_1\ ) \\ &\qquad\qquad + x_1\cdot(\ x_1\cdot(x_2)^2 - x_1\cdot(x_2)^2\ ) \\ &= 0. \end{aligned}$$

Since the matrix is singular, it is positive semi-definite. At this point, the test for a global minimum is inconclusive, since it is entirely possible that the second partial evaluated at the optimum weights may be a saddle point. Therefore, it is necessary to force the matrix to be positive definite and insure a global minimum by adding a scaled quadratic function of the weights to s(**w**), such that

$$S(\mathbf{w}) = s(\mathbf{w}) + \epsilon\cdot(\ \mathbf{w} - \mathbf{w}^*\ )^T\cdot(\ \mathbf{w} - \mathbf{w}^*\ ),\quad \epsilon > 0. \qquad \text{D. 4}$$

This particular quadratic was chosen, such that when the first partial derivative is taken and evaluated at **w***, the partial of the quadratic reduces to zero under ideal conditions. The second partial of S(**w**) evaluated at the set of optimum weights (**w***) is

$$\frac{\partial^2 S(\mathbf{w}^*)}{\partial \mathbf{w} \partial \mathbf{w}^T} = \frac{\partial^2 s(\mathbf{w}^*)}{\partial \mathbf{w} \partial \mathbf{w}^T} + 2 \cdot \epsilon \cdot \mathbf{I} > 0$$

and positive definite if

$$f(\mathbf{x}^j, \mathbf{w}) \cdot (1 - f(\mathbf{x}^j, \mathbf{w})) - (d - f(\mathbf{x}^j, \mathbf{w})) \cdot (1 - 2 \cdot f(\mathbf{x}^j, \mathbf{w})) \geq 0, \quad \text{for each } j.$$

Two cases must be considered, d = 1 and d = 0. For d = 0, a new function of the output may be described, such that

$$G_0(f) = f \cdot (2 - 3 \cdot f)$$

for a particular input and a given set of weights. The graph of $G_0(f)$ is shown below in Fig. D.2. Keep in mind that the values of the sigmoid function are continuous over the range (0, 1). This implies that $G_0(f)$ is non-negative over $0 \leq f \leq 2/3$.

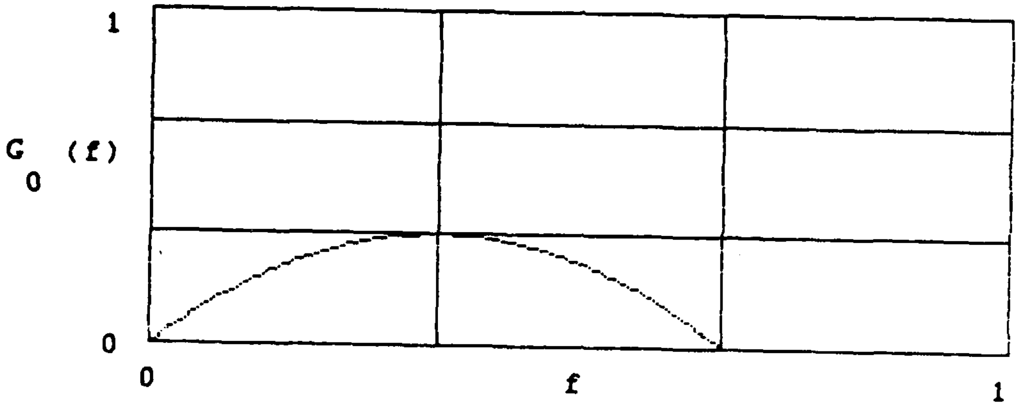


Figure D.2 Quadratic Function for d = 0

When d = 1, the function takes on the following form:

$$G_1(f) = -3 \cdot f^2 + 4 \cdot f - 1$$

The quadratic equation provides the points where the function crosses the zero axis, such that $G_1(f)$ is non-negative over the range $1/3 \leq f \leq 1$. The graph of $G_1(f)$ is shown below in Fig. D.3.

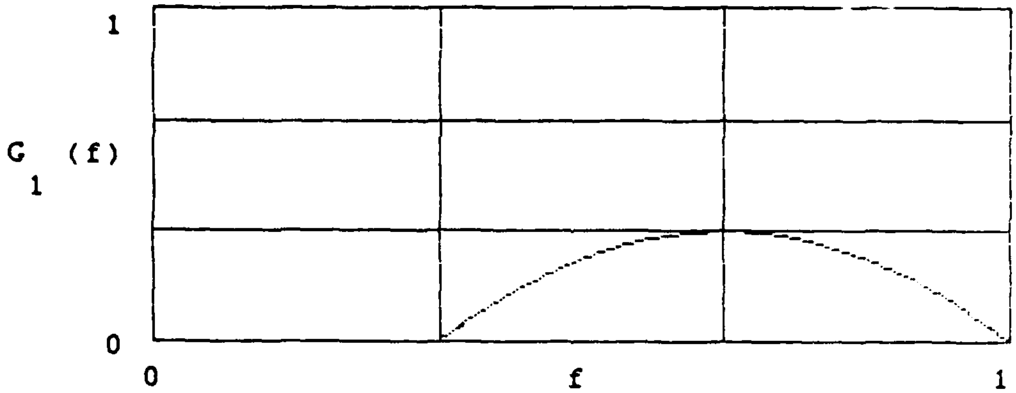


Figure D.3 Quadratic Function for d = 1

By over lapping the two graphs of Figs. D.2 and D.3, the range of values f can assume is $1/3 \le f \le 2/3$. If the output meets this initial criterion, then the second partial of the squared error is a positive definite matrix. The criterion placed on the initial weight values, such that the training begins in the neighborhood of the global minimum can be found by rewriting this inequality. This is accomplished below.

Consider the range of values the output of the sigmoid function may take on, in order to begin training within the neighborhood of a global minimum; such that

$$\frac{1}{3} \le f(x^T \cdot w) \le \frac{2}{3}$$

$$\frac{1}{3} \le \frac{1}{1 + \exp(-x^T \cdot w)} \le \frac{2}{3}$$

$$1 \le \frac{3}{1 + \exp(-x^T \cdot w)} \le 2$$

$$1 + \exp(-x^T \cdot w) \le 3 \le 2 + 2 \cdot \exp(-x^T \cdot w).$$

Consider the lower bound, where

$$\exp(-x^T \cdot w) \le 2.$$

The lower bound for the weighted sum of the inputs becomes:

$$x^T \cdot w \ge -\ln(2).$$

The upper bound is found in a similar fashion,

$$\exp(-\mathbf{x}^T \cdot \mathbf{w}) \geq 1$$

$$\mathbf{x}^T \cdot \mathbf{w} \leq \ln(2).$$

The final criterion becomes,

$$-\ln(2) \leq \mathbf{x}^T \cdot \mathbf{w} \leq \ln(2).$$

This relationship must hold over the entire input vector ensemble. If this criterion is met, then it is insured that training will begin with a set of weight values in the neighborhood of a global minimum over the entire input ensemble. Therefore, when considering the entire input training set,

$$-\ln(2) \leq (\mathbf{x}^j)^T \cdot \mathbf{w} \leq \ln(2). \qquad \text{D. 5}$$

Results of this relationship suggest that the optimal weight values are bounded in weight space by hyperplanes. These hyperplanes are described from the above criterion, if $\mathbf{x}^T \cdot \mathbf{w}$ is allowed to equal the two extremes, -ln(2) and ln(2), for a specific input. The result is a hypercube in weight space. The ensemble of hypercubes over all input vectors approximates a sphere in weight space enclosing the optimal weight values for the corresponding set of input vectors.

By placing some restrictions on the magnitude of the input vectors and then analyzing the criterion of Eq. D.5, the initial range of weight values may be randomly chosen. For instance, assume that the inputs have been normalized to lie in the interval (-1, 1). By restating Eq. D.5 in the following manner:

$$-\ln(2) \le \sum_{i=1}^{n} w_i \cdot x_i \le \ln(2),$$

where n ranges over all inputs to the cell. With the above inequality, consider a worst case condition; assume that all the inputs are all equal to 1 (or -1). This condition places a further restriction on the initial value of the weights, than imposed by the above inequality of Eq. D.5, such that

$$-\ln(2) \le \sum_{i=1}^{n} w_i \le \ln(2)$$

and

$$-\ln(2) \le \sum_{i=1}^{n} w_i \le \ln(2). \qquad \text{D. 6}$$

The above inequality provides a strict criterion for the initial weight values of each cell. For applications within this study, the weights are randomly set. It would be a trivial exercise to perform the above criterion and reset the weights of those cells which do not meet the inequality.

Two important results should be observed from the discussion above. First, if the criterion of Eq. D.6 is met, then a the inequality of D.5 is met. This implies that the second partial derivative of the squared error is a positive definite matrix.

The existence of the positive definite matrix implies the existence of a global minimum over the corresponding input set. Furthermore, training begins within some neighborhood of the global minimum. Secondly, randomly setting the weights to very small, negative and positive, values provides a near zero value for the sum of weighted inputs. This implies that all output nodes fire on average, near 0.5, allowing the network to train and drive the outputs toward desired values. If the nodal outputs are driven towards extremely low or high values initially, the network has a very difficult task of driving the network towards desired values in the opposite direction.

D.2. The Momentum Term

The weighted quadratic function added to the squared error term, is an attempt at driving the second partial matrix of the squared error to a positive definite matrix. However, many researchers desire to use this quadratic, with the idea of enhancing convergence times. For instance, consider Eq. D.4 as the function to minimize during training. Reproducing Eq. D.4 provides,

$$S(\mathbf{w}) = \frac{1}{k} \cdot \sum_{j=1}^{k} (d - f(x^{j}, \mathbf{w}))^{2} + \epsilon \cdot (\mathbf{w} - \mathbf{w}^{*})^{T} \cdot (\mathbf{w} - \mathbf{w}^{T}).$$

Using a first order technique, the weights are changing according to the following first order differential equation:

$$\frac{\partial \mathbf{w}}{\partial t} = \frac{\partial}{\partial \mathbf{w}}\left(-\beta \cdot s + \epsilon \cdot (\ \mathbf{w} - \mathbf{w}^{*}\ )^{T} \cdot (\ \mathbf{w} - \mathbf{w}^{*} \right).$$

Lippman describes the discrete counterpart of the above differential equation [4:17]. By computing the partial, the weights are updated by:

$$\frac{\partial \mathbf{w}}{\partial t} = \frac{2 \cdot \beta}{k} \cdot \sum_{j=1}^{k} (\ d - f(\mathbf{x}^{j},\ \mathbf{w})\ ) \cdot f(\mathbf{x}^{j},\ \mathbf{w}) \cdot (\ 1 - f(\mathbf{x}^{j},\ \mathbf{w})\ ) \cdot \mathbf{x}^{j}$$
$$+ 2 \cdot \epsilon \cdot (\ \mathbf{w} - \mathbf{w}^{*}\ )$$

The parameter β controls the convergence rate. The last term, $2 \cdot \epsilon \cdot (\mathbf{w} - \mathbf{w}^{*})$, is known as the momentum term and first introduced by Rummelhart, Hinton, and Williams [13]. An ariticle by Lippman describes the momentum term as possibly improving convergence times [4:17]. In the derivation of the second order approximation, Parker introduces the momentum term by means of leakage terms [7:593-600; 8]. Parker believes the leakage terms insure convergence.

In the last section, conditions were established such that the network could begin training with a second order algorithm. However, what happens to the positive definite matrix of the second partial of S as the network trains? To maintain the status of a positive definite matrix, there must be some condition placed on ϵ. In this section, it is desired to find the optimal momentum scalar, ϵ, insuring that the search is always being performed near the global minimum.

Consider Eq. D.4, which defines a new performance surface, where

$$S(\mathbf{w}) = s(\mathbf{w}) + \epsilon \cdot (\mathbf{w} - \mathbf{w}^{*})^{T} \cdot (\mathbf{w} - \mathbf{w}^{*})$$

and the average second partial of S becomes:

$$\frac{\partial^{2} S(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^{T}} = \frac{2}{k} \cdot \sum_{j=1}^{k} f(\mathbf{x}^{j}, \mathbf{w}) \cdot (1 - f(\mathbf{x}^{j}, \mathbf{w})) \cdot \Big\{ f(\mathbf{x}^{j}, \mathbf{w}) \cdot (1 - f(\mathbf{x}^{j}, \mathbf{w})) - (d - f(\mathbf{x}^{j}, \mathbf{w})) \cdot (1 - 2 \cdot f(\mathbf{x}^{j}, \mathbf{w})) \Big\} \cdot \mathbf{x}^{j} \cdot (\mathbf{x}^{j})^{T} + 2 \cdot \epsilon \cdot \mathbf{I}.$$

Let

$$\alpha(\mathbf{x}^{j}, \mathbf{w}) = f(\mathbf{x}^{j}, \mathbf{w}) \cdot (1 - f(\mathbf{x}^{j}, \mathbf{w})) \cdot \Big\{ f(\mathbf{x}^{j}, \mathbf{w}) \cdot (1 - f(\mathbf{x}^{j}, \mathbf{w})) - (d - f(\mathbf{x}^{j}, \mathbf{w})) \cdot (1 - 2 \cdot f(\mathbf{x}^{j}, \mathbf{w})) \Big\},$$

then

$$\frac{\partial^{2} S(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^{T}} = \frac{2}{k} \cdot \sum_{j=1}^{k} \alpha(\mathbf{x}^{j}, \mathbf{w}) \cdot \mathbf{x}^{j} \cdot (\mathbf{x}^{j})^{T} + 2 \cdot \epsilon \cdot \mathbf{I}.$$

Given an input pattern and set of weights α becomes a scalar. Its dependence on $\mathbf{x}^{j}$ and $\mathbf{w}$ will be removed for convenience.

To remain within the global minimum, the matrix above must maintain its positive definite condition. Again, the test for a positive definite matrix is applied, such that

$$\mathbf{y}^T\cdot(\ \alpha\cdot\mathbf{x}^j\cdot(\mathbf{x}^j)^T + 2:\epsilon:\mathbf{I}\ )\cdot\mathbf{y} > 0,$$

for all $j = 1, 2, \ldots, k$ and for all $\mathbf{y}$ which are members of $\mathbf{R}^n$. So

$$(\ \alpha\cdot\mathbf{y}^T\cdot\mathbf{x}^j\cdot(\mathbf{x}^j)^T:\mathbf{y} + 2:\epsilon:\mathbf{y}^T\cdot\mathbf{y}\ ) =$$

$$(\ \alpha\cdot(\ \mathbf{y}^T\cdot\mathbf{x}^j\ )\cdot(\ (\mathbf{x}^j)^T:\mathbf{y}\ ) + 2:\epsilon:\mathbf{y}^T\cdot\mathbf{y}\ ) =$$

$$(\ \alpha\cdot(\ (\mathbf{x}^j)^T\cdot\mathbf{y}\ )^T\cdot(\ (\mathbf{x}^j)^T:\mathbf{y}\ ) + 2:\epsilon:\mathbf{y}^T\cdot\mathbf{y}\ )$$

that is, we wish

$$\alpha\cdot(\ (\mathbf{x}^j)^T\cdot\mathbf{y}\ )^2 + 2\cdot\epsilon\cdot\mathbf{y}^T\cdot\mathbf{y} > 0 \qquad \text{D. 7}$$

Two cases must be considered along the way to insure that the above inequality is met. First, given an $\alpha > 0$ and an $\mathbf{x}^j$, is there a condition on ϵ such that the inequality is met? Yes, $\epsilon > 0$, where ϵ is independent of α and $\mathbf{x}^j$. The second case is not so trivial. For, given an $\alpha < 0$ and an $\mathbf{x}^j$, what condition is placed on ϵ, where $\epsilon = \epsilon(\alpha, \mathbf{x}^j)$? Rewriting Eq. D.7,

$$\epsilon > \frac{-\alpha\cdot(\ (\mathbf{x}^j)^T\cdot\mathbf{y}\ )^2}{2\cdot\mathbf{y}^T\cdot\mathbf{y}}. \qquad \text{D. 8}$$

Recall that $\mathbf{y}$ is a nonzero vector. Figures D.4 and D.5 are plots of α as a function of the output (f) for values of d = 0 and d = 1 respectively.

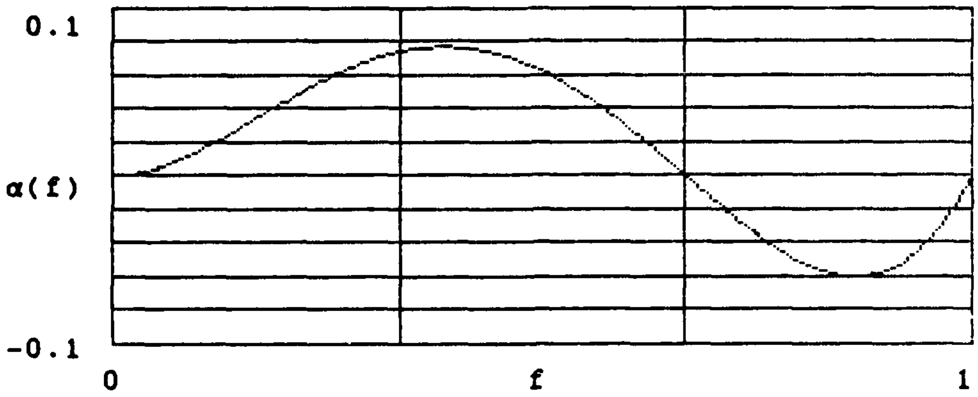


Figure D.4 α(f) for d = 0

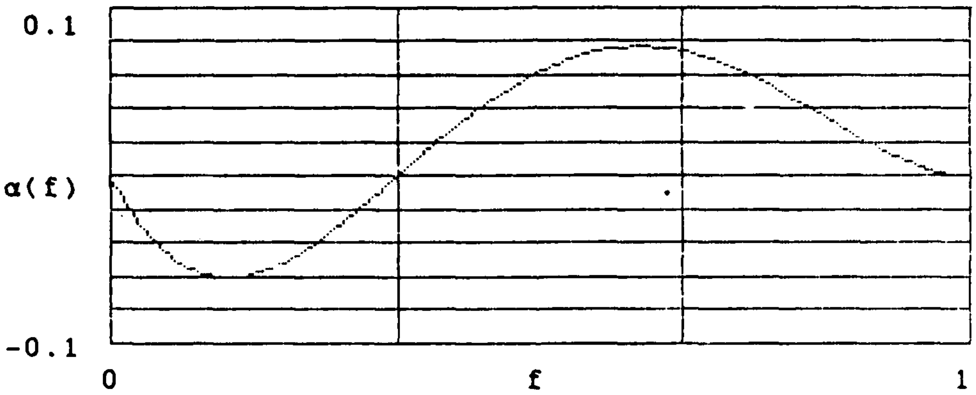


Figure D.5 α(f) for d=1

For a value of ϵ independent of $\mathbf{x}^J$ and α, the most negative value of α is desired. From Figs. D.4 and D.5 that value is α = -0.06. It is also desired to have the maximum eigenvalue, such that

$$\mathbf{A} = \frac{-\alpha \cdot ((\mathbf{x}^J)^T \cdot \mathbf{y})^2}{\mathbf{y}^T \cdot \mathbf{y}}$$

and

$$\mathbf{A} = \lambda_{max}$$

where the eigenvalues are found by

$$\mathbf{y} \cdot \left| \mathbf{x}^{j} \cdot (\mathbf{x}^{j})^{T} - \lambda \cdot \mathbf{I} \right| \cdot \mathbf{y}^{T} = 0$$

or

$$\begin{vmatrix} x_1 \cdot x_1 - \lambda & x_1 \cdot x_2 & x_1 \\ x_2 \cdot x_1 & x_2 \cdot x_2 - \lambda & x_2 \\ x_1 & x_2 & 1 - \lambda \end{vmatrix} = 0$$

for a particular input vector j. The above determinant produces a cubic in λ, where

$$\lambda^{2} \cdot (-\lambda + (x_1^{2} + x_2^{2} + 1)) = 0$$

and

$$\lambda_{max} = x_1^{2} + x_2^{2} + 1.$$

Finally, ϵ is estimated as:

$$\epsilon > \frac{1}{k} \cdot \sum_{j=1}^{k} \alpha(\mathbf{x}^{j}, \mathbf{w}) \cdot ((x_1^{j})^{2} + (x_2^{j})^{2} + 1).$$

This expression is for the specific problem described above in section D.1. In general, while considering the minimum α (-0.06), ϵ can be rewritten as:

$$\epsilon > 0.06 \cdot \frac{1}{k} \cdot \sum_{j=1}^{k} \left(\left(\sum_{i=1}^{n} (x_i^j)^2 \right) + 1 \right) \qquad \text{D. 9}$$

From the above inequality, ϵ is a function of the sum of the square of the input components averaged over the entire ensemble of input vectors. The input equal to 1 is analogous to a threshold.

By adding the quadratic function of the weights to the function (squared error) being minimized, a smoother error surface results. It is believed the quadratic will (for lack of a better expression) stretch out inflection and/or saddle points, providing a smoother upward concavity. If the region in proximity of the global minimum is relatively flat, the quadratic will increase convergence times, again by providing an upward concavity. Thus the quadratic removes areas within the error surface which may slow down the converging process. The expression for ϵ in Eq. D.9, insures that this condition is maintained throughout training.

One question remains to be answered. From the above discussions, **w*** is the optimum vector of weights used to determine the minimum of the error surface. So what value is used to approximate **w***? - It's the best estimate of the previous weight values, approximated by the weight update rule being used.

Appendix E: Further Comparisons with the Bayesian Classifier

This appendix provides an extended comparison of the neural net classifier and Bayesian classifier. However, the criterion for determining correct classification has been altered for the neural net classifier. In chapter five, the node corresponding to a correct choice, had to fire above 0.8, while all other nodes fire less 0.2. This criterion will be eased a bit to provide a comparable analysis (if possible). Now, the node corresponding to a correct classification must fire above 0.5 and all other nodes below 0.5. Table E.1 provides the results for a single pass through the network.

Table E.1 Overall Classification Accuracy. (1) Gradient of Steepest Descent, (2) Momentum Method, (3) Second Order Method, (4) Bayesian.

Overall Accuracy				
	(1)	(2)	(3)	(4)
Training Data	**	91.9%	88.7%	74.8%
Testing Data	**	67.8%	72.4%	75.3%

The above table represents an instance during a typical training session. Again, the neural net classifiers far exceed the performance levels of the Bayesian classifier, when the training data is considered. However, the Bayesian classifier has a sizable edge on the momentum method and a slight edge on the second order method, when regarding the test data. It is likely that the neural net classifier would improve, if the amount of information is increased. For instance, by increasing the number of original input features. The neural net learns by example, the more information the net has, the more opportunity the net has to learn it's environment.

Appendix F: XOR Model

```
with text_io;                    use text_io;
with integer_text_io;            use integer_text_io;
with float_text_io;              use float_text_io;
with float_math_lib;             use float_math_lib;
with system;

with MATH_LIB_EXTENSION;         use MATH_LIB_EXTENSION;
with VECTOR_OPERATIONS;          use VECTOR_OPERATIONS;
with somp_support;               use somp_support;

procedure SO_XOR is

num_inputs            : integer;
num_L1_nodes          : integer;
num_L2_nodes          : integer;
A1                    : float;
A2                    : float;
A3                    : float;
A4                    : float;
A5                    : float;

total_error           : float;
Iteration_Count       : integer  := 0;
Convergence_Count     : integer  := 0;
Convergence_Criterion : constant := 0.1;
interval              : integer;

Center                : float;
Width                 : float;
total_cost            : float;

Seed                  : system.unsigned_longword := MATH_LIB_EXTENSION.get_seed;

begin  --Main

put ("Enter center of random weight distribution: "); get (center);
  skip_line;
put ("Enter width of random weight distribution: "); get (width);
  skip_line;

put ("Enter number of inputs: "); get (num_inputs); skip_line;
put ("Enter number of L1 nodes: "); get (num_L1_nodes); skip_line;
put ("Enter number of L2 nodes: "); get (num_L2_nodes); skip_line;
put ("Enter interval: "); get (interval); skip_line;
put ("Enter the constant A1: "); get (A1); skip_line;
put ("Enter the constant A2: "); get (A2); skip_line;
put ("Enter the constant A3: "); get (A3); skip_line;
put ("Enter the constant A4: "); get (A4); skip_line;
put ("Enter the constant A5: "); get (A5); skip_line;

declare

L1   : layer ( num_inputs, num_L1_nodes );
L2   : layer ( num_L1_nodes, num_L2_nodes );
Dout : vector ( 1 .. num_L2_nodes );

begin

--Initialize network parameters

for j in L1.W'range(2) loop
  for i in L1.W'range(1) loop
    uniform ( center, width, seed, L1.W(i, j) );
    L1.Del_W(i, j) := 0.0;
  end loop;
  uniform ( center, width, seed, L1.Theta(j) );
  L1.Del_Theta(j) := 0.0;
end loop;

for j in L2.W'range(2) loop
  for i in L2.W'range(1) loop
    uniform ( center, width, seed, L2.W(i, j) );
    L2.Del_W(i, j) := 0.0;
  end loop;
  uniform ( center, width, seed, L2.Theta(j) );
  L2.Del_Theta(j) := 0.0;
end loop;

while Convergence_Count /= 4 loop

  if Iteration_Count mod 2 = 0 then
    L1.Fin(1) := 0.1;
  else
    L1.Fin(1) := 0.9;
  end if;

  if Iteration_Count mod 4 < 2 then
    L1.Fin(2) := 0.1;
  else
  L1.Fin(2) := 0.9;
end if;

if Iteration_Count mod 4 = 0 or Iteration_Count mod 4 = 3 then
  Dout(1) := 0.1;
else
  Dout(1) := 0.9;
end if;

forward_pass (L1, A3, A5 );

L2.Fin       := L1.Fout;
L2.Fin_Prime := L1.Fout_Prime;

forward_pass ( L2, A3, A5 );

L2.Etotal := 2.0 * ( Dout - L2.Fout );

for i in L2.Fout_Prime'range loop
  L2.Etotal_Prime(i) := -2.0 * L2.Fout_Prime(i);
end loop;

total_error := sum_output_error ( L2.Etotal );

if abs ( total_error ) < Convergence_Criterion then
  Convergence_Count := Convergence_Count +1;
else
  Convergence_Count := 0;
end if;

backward_pass ( L2, A3, A5 );

L1.Etotal := compute_sum ( L2.Eout );
L1.Etotal_Prime := compute_sum ( L2.Eout_Prime );

backward_pass ( L1, A3, A5 );

total_cost := 0.0;

update_weights ( L2, total_cost, A1, A2, A4 );
update_weights ( L1, total_cost, A1, A2, A4 );

update_thresholds ( L2, total_cost, A1, A2, A4 );
update_thresholds ( L1, total_cost, A1, A2, A4 );
```

```
  Iteration_Count := Iteration_Count + 1;

  if Iteration_Count mod interval = 0 then
    new_line;
    for i in L2.Fout'range loop
      put ("Error = "); put (L2.Etotal(i)/2.0); put ( "  " );
    end loop;
    put ("Iteration = "); put (Iteration_Count); new_line(2);
 end if;

end loop;

put ("Iterations till Convergence = "); put (Iteration_Count);

end;

end SO_XOR;
```

```
--****************************************************************
--*                                                              *
--* Appendix G: ADA Programming Model                            *
--*                                                              *
--* This program is a computer simulation of a biological-based  *
--* neural network, applying a modified backward error propaga-  *
--* tion (BEP) algorithm.  This neural network model was         *
--* developed for applications in pattern classification.  The   *
--* modified BEP uses a minimization technique based on an       *
--* approximation to a second order Newton's method.  This       *
--* algorithm takes advantage of second order derivatives (of    *
--* the surface to be minimized), as well as first order deriva- *
--* tives.  Time derivatives of the signals propagating through  *
--* the network are also used in updating the network weights.  *
--* Below is the main procedure, Second Order Multilayer         *
--* Perceptron (SOMP1.ADA) written in the ADA programming        *
--* environment.                                                 *
--*                                                              *
--* Model implemented by: Capt Clark Piazza, USAF                *
--*                                                              *
--****************************************************************


with system;
with text_io;                    use text_io;
with float_text_io;              use float_text_io;
with integer_text_io;            use integer_text_io;
with float_math_lib;             use float_math_lib;

with somp_io;                    use somp_io;
with somp_support;               use somp_support;
with vector_operations;          use vector_operations;
with math_lib_extension;         use math_lib_extension;

procedure somp1 is

center             : float;
width              : float;
num_L1_nodes       : integer;
num_L2_nodes       : integer;
num_classes        : integer;
num_tr_patterns    : integer;
num_te_patterns    : integer;
num_moments        : integer;
A1                 : float;
A2                 : float;
A3                 : float;
A4                 : float;
A5                 : float;

-- Testing and training files containing pattern feature vectors.
-- The file containing the features, must of type string 16
-- characters long.
```

```
te_list              : string ( 1 .. 16 );
tr_list              : string ( 1 .. 16 );

seed : system.unsigned_longword := math_lib_extension.get_seed;

output_error         : float;
error_tolerance      : float;
total_cost           : float;
total_distance       : float;
avg_distance         : float;
max1_index           : integer;
max2_index           : integer;
max_iterations       : integer;
interval             : integer;
num_iterations       : integer;
te_count             : integer;
tr_count             : integer;
tr_num_correct       : float;
te_num_correct       : float;

tr_accuracy          : float;
te_accuracy          : float;
tot_tr_error         : float;
tot_te_error         : float;
avg_err_per_pat      : float;
num_passes           : integer := 1;
tot_passes           : integer;
num_points           : integer;

convergence          : boolean;

tr_error_file        : text_io.file_type;
te_error_file        : text_io.file_type;
tr_accuracy_file     : text_io.file_type;
te_accuracy_file     : text_io.file_type;


-- Begin main procedure.

begin

-- Enter the following from the terminal or create a com file.

put ( "Enter center of random weight distribution (center),
 type float: " ); get ( center ); skip_line;
put ( "Enter width of random weight distribution (width),
 type float: " ); get ( width ); skip_line;
put ( "Enter number of layer one nodes (num_L1_nodes),
 type integer: " ); get ( num_L1_nodes ); skip_line;
put ( "Enter number of layer two nodes (num_L2_nodes),
 type integer: " ); get ( num_L2_nodes ); skip_line;
put ( "Enter number of output nodes (num_classes),
 type integer: " ); get ( num_classes ); skip_line;
put ( "Enter number of training patterns (num_tr_patterns),
 type integer: " ); get ( num_tr_patterns ); skip_line;
put ( "Enter number of testing patterns (num_te_patterns),
 type integer: " ); get ( num_te_patterns ); skip_line;
put ( "Enter number of moments per pattern (num_moments),
 type integer: " ); get ( num_moments ); skip_line;
put ( "Enter training moment data file (tr_list),
 type string: " ); get ( tr_list ); skip_line;
put ( "Enter testing moment data file (te_list),
 type string: " ); get ( te_list ); skip_line;
put ( "Enter number of separate training passes (tot_passes),
 type integer: " ); get ( tot_passes ); skip_line;
put ( "Enter maximum number of iterations (max_iterations),
 type integer: " ); get ( max_iterations ); skip_line;
put ( "Enter output interval to examine results,
 type integer: " ); get ( interval ); skip_line;
put ( "Enter error tolerance (error_tolerance),
 type float: " ); get ( error_tolerance ); skip_line;

-- Enter the desired learning parameters.  A1 controls convergence
-- for first order method.  A2 and A3 control the amount of noise
-- induced into the network.  A4 controls the amount of momentum.
-- A5 is a convergence term controlling the second derivative
-- information.

put ( "Enter constant A1, type float: " ); get ( A1 ); skip_line;
put ( "Enter constant A2, type float: " ); get ( A2 ); skip_line;
put ( "Enter constant A3, type float: " ); get ( A3 ); skip_line;
put ( "Enter constant A4, type float: " ); get ( A4 ); skip_line;
put ( "Enter constant A5, type float: " ); get ( A5 ); skip_line;

create ( tr_error_file, out_file, "tr_error.dat" );
create ( te_error_file, out_file, "te_error.dat" );
create ( tr_accuracy_file, out_file, "tr_accuracy.dat" );
create ( te_accuracy_file, out_file, "te_accuracy.dat" );


-- Declare network layer variables.

declare

L1              : layer ( num_moments, num_L1_nodes );
L2              : layer ( num_L1_nodes, num_L2_nodes );
L3              : layer ( num_L2_nodes, num_classes );
Dout            : vector ( 1 .. num_classes );
training_array  : matrix ( 1 .. num_tr_patterns, 1 .. num_moments + 1 );
testing_array   : matrix ( 1 .. num_te_patterns, 1 .. num_moments + 1 );

-- Interval must be some multiple of max_iterations.

data_points     : integer := ( max_iterations / interval ) + 1;

avg_tr_error    : vector ( 1 .. data_points );
avg_te_error    : vector ( 1 .. data_points );
avg_tr_acc      : vector ( 1 .. data_points );
```

```
avg_te_acc     : vector ( 1 .. data_points );
tr_acc_array   : matrix ( 1 .. tot_passes, 1 .. data_points );
te_acc_array   : matrix ( 1 .. tot_passes, 1 .. data_points );
tr_err_array   : matrix ( 1 .. tot_passes, 1 .. data_points );
te_err_array   : matrix ( 1 .. tot_passes, 1 .. data_points );

-- The parameters below may be used to measure the amount of change
-- of the weights and thresholds, via cost.

L1_weights     : constant natural := num_moments * num_L1_nodes;
L1_thresholds  : constant natural := num_L1_nodes;
L2_weights     : constant natural := num_L1_nodes * num_L2_nodes;
L2_thresholds  : constant natural := num_L2_nodes;
L3_weights     : constant natural := num_L2_nodes * num_classes;
L3_thresholds  : constant natural := num_classes;

tot_parameters : constant float :=
                             float( L1_weights + L1_thresholds +
                                    L2_weights + L2_thresholds +
                                    L3_weights + L3_thresholds );

-- Begin declare block.

begin

-- Get training and testing moments, store into an array.

get_moment_array ( tr_list, num_moments, training_array );
get_moment_array ( te_list, num_moments, testing_array );

-- Initialize and train network a predetermined number of times
-- and average network performance.

while num_passes <= tot_passes loop

-- Initialize network variables.

  initialize_network ( L1, L2, L3, center, width, seed );

-- Begin training.

  num_points          := 1;
  num_iterations      := 0;
  convergence         := false;

 while                     num_iterations <= max_iterations loop
     -- convergence = false and

    generate_random_moms ( num_tr_patterns, num_moments,
                     training_array, L1.Fin, Dout, seed );

  -- Begin forward pass through network.
```

```
   compute_forward_pass ( L1, L2, L3, A3, A5 );

 -- Compute output error and time derivative of error for each
 -- output node.

   L3.Etotal := 2.0 * ( Dout - L3.Fout );

   for i in L3.Fout_Prime'range loop
     L3.Etotal_Prime(i) := -2.0 * L3.Fout_Prime(i);
   end loop;

 -- Compute sum of output errors, sum and average over all
 -- input patterns, and test with error tolerance for convergence.

-- output_error := sum_output_error ( L3.Etotal );
-- if num_iterations mod num_tr_patterns /= 0
--                      or num_iterations = 0 then
-- avg_err_per_pat := avg_err_per_pat
--     + ( output_error / float(num_tr_patterns ) );
-- elsif avg_err_per_pat < error_tolerance then
-- convergence := true;
-- else
-- convergence := false;
-- end if;

-- Begin backward pass through the network one layer at a time.

   backward_pass ( L3, A3, A5 );

   L2.Etotal       := compute_sum ( L3.Eout );
   L2.Etotal_Prime := compute_sum ( L3.Eout_Prime );

   backward_pass ( L2, A3, A5 );

   L1.Etotal       := compute_sum ( L2.Eout );
   L1.Etotal_Prime := compute_sum ( L2.Eout_Prime );

   backward_pass ( L1, A3, A5 );

   total_cost := 0.0;

   update_weights ( L3, total_cost, A1, A2, A4 );
   update_weights ( L2, total_cost, A1, A2, A4 );
   update_weights ( L1, total_cost, A1, A2, A4 );

   update_thresholds ( L3, total_cost, A1, A2, A4 );
   update_thresholds ( L2, total_cost, A1, A2, A4 );
   update_thresholds ( L1, total_cost, A1, A2, A4 );

   -- Used to measure network parameter changes.

   -- avg_distance := sqrt( total_cost ) / tot_parameters
-- Compute network performance.

if num_iterations mod interval = 0 then

-- Check training data performance.

  tot_tr_error   := 0.0;
  tr_num_correct := 0.0;
  tr_count := 1;

  while tr_count <= num_tr_patterns loop

    generate_seq_moms ( tr_count, num_moments, training_array,
                        L1.Fin, Dout );

    compute_forward_pass ( L1, L2, L3, A3, A5 );

    find_max_vals ( L3.Fout, max1_index, max2_index );

    tr_num_correct := tr_num_correct
      + float ( correct ( max1_index, max2_index,
                          L3.Fout, Dout ) );

    L3.Etotal := 2.0 * ( Dout - L3.Fout );

    output_error := sum_output_error ( L3.Etotal );

    tot_tr_error := tot_tr_error +  output_error;

    tr_count := tr_count + 1;

  end loop;

  tr_acc_array ( num_passes, num_points )
    := compute_ratio ( tr_num_correct, num_tr_patterns );

  tr_err_array ( num_passes, num_points )
    := compute_ratio ( tot_tr_error, num_tr_patterns );

  -- Check test data performance.

  tot_te_error   := 0.0;
  te_num_correct := 0.0;
  te_count       := 1;

  while te_count <= num_te_patterns loop

    generate_seq_moms ( te_count, num_moments, testing_array,
                         L1.Fin, Dout );

    compute_forward_pass ( L1, L2, L3, A3, A5 );

    find_max_vals ( L3.Fout, max1_index, max2_index );
```

```
          te_num_correct := te_num_correct
            + float ( correct ( max1_index, max2_index,
                                L3.Fout, Dout ) );

          L3.Etotal := 2.0 * ( Dout - L3.Fout );

          output_error := sum_output_error ( L3.Etotal );

          tot_te_error := tot_te_error + output_error;

          te_count := te_count + 1;

        end loop;

        te_acc_array ( num_passes, num_points )
          := compute_ratio ( te_num_correct, num_te_patterns );

        te_err_array ( num_passes, num_points )
          := compute_ratio ( tot_te_error, num_te_patterns );

        num_points := num_points + 1;

      end if;

      num_iterations := num_iterations + 1;

    end loop;

    num_passes := num_passes + 1;

  end loop;

  -- Compute average network performance.

  avg_tr_error    := compute_average ( tr_err_array, tot_passes );
  avg_te_error    := compute_average ( te_err_array, tot_passes );
  avg_tr_acc      := compute_average ( tr_acc_array, tot_passes );
  avg_te_acc      := compute_average ( te_acc_array, tot_passes );

  -- Store average network performance in matrixX format.

  store_net_perf ( avg_tr_error, tr_error_file, interval );
  store_net_perf ( avg_te_error, te_error_file, interval );
  store_net_perf ( avg_tr_acc, tr_accuracy_file, interval );
  store_net_perf ( avg_te_acc, te_accuracy_file, interval );

  -- Close all files.

  close ( tr_error_file );
  close ( te_error_file );
  close ( tr_accuracy_file );
  close ( te_accuracy_file );
```

```
-- End declare block.
end;
-- End main procedure.
end sompl;
```

```
with system;
with text_io;                  use text_io;

with vector_operations;        use vector_operations;
with math_lib_extension;       use math_lib_extension;

package somp_support is

type layer ( inputs : positive; outputs : positive ) is
 record

 -- Network Parameters

 Fin           : vector ( 1 .. inputs );
 Fin_Prime     : vector ( 1 .. inputs );
 W             : matrix ( 1 .. inputs, 1 .. outputs );
 Del_W         : matrix ( 1 .. inputs, 1 .. outputs );
 Theta         : vector ( 1 .. outputs );
 Del_Theta     : vector ( 1 .. outputs );
 Fout          : vector ( 1 .. outputs );
 Fout_Prime    : vector ( 1 .. outputs );
 Eout          : matrix ( 1 .. inputs, 1 .. outputs );
 Eout_Prime    : matrix ( 1 .. inputs, 1 .. outputs );
 Etotal        : vector ( 1 .. outputs );
 Etotal_Prime  : vector ( 1 .. outputs );

 -- Temporary variables

 X             : vector ( 1 .. outputs );
 V             : vector ( 1 .. outputs );
 U             : vector ( 1 .. outputs );
 Q             : vector ( 1 .. outputs );
 R             : vector ( 1 .. outputs );

end record;


function sigmoid ( input : vector ) return vector;

function compute_sum ( input : matrix ) return vector;

function sum_output_error ( input : vector ) return float;

function correct ( index1, index2  : integer;
                   output, desired : vector ) return integer;

function compute_ratio ( numerator   : float ;
                         denominator : integer ) return float;

function compute_average ( perf_array : matrix;
                         tot_passes : integer ) return vector;
```

```
procedure initialize_network ( L1, L2, L3 : in out layer;
              center     : in float;
              width      : in float;
              seed       : in out system.unsigned_longword );

procedure forward_pass ( L      : in out layer;
                         A3, A5 : in float );

procedure compute_forward_pass ( L1, L2, L3 : in out layer;
                                 A3, A5     : in float );

procedure backward_pass ( L      : in out layer;
                          A3, A5 : in float );

procedure update_weights ( L          : in out layer;
                           cost       : in out float;
                           A1, A2, A4 : in float );

procedure update_thresholds ( L          : in out layer;
                              cost       : in out float;
                              A1, A2, A4 : in float );

procedure find_max_vals ( output : in vector;
                          index1 : in out integer;
                          index2 : in out integer );

end somp_support;
```

```
with float_math_lib;        use float_math_lib;

package body somp_support is

function sigmoid ( input : vector ) return vector is

output : vector ( input'range );

begin

for i in input'range loop
  begin
    output(i) := 1.0 / ( 1.0 + exp ( -input(i) ) );
  exception
    when FLOOVEMAT => output(i) := 0.0;
  end;
end loop;

return output;

end sigmoid;


function compute_sum ( input : matrix ) return vector is

total : vector ( input'range(1) ) := ( others => 0.0 );

begin

for i in input'range(1) loop
  for j in input'range(2) loop
    total(i) := total(i) + input(i, j);
  end loop;
end loop;

return total;

end compute_sum;


function sum_output_error ( input : vector ) return float is

total : float := 0.0;

begin

for i in input'range loop
  total := ( total + abs( input(i) ) ) / 2.0;
end loop;

return total;

end sum_output_error;
```

```
function correct ( index1, index2  : integer;
             output, desired : vector ) return integer is

update : integer;

begin

if output(index1) >= 0.5 and output(index2) < 0.5
                         and desired(index1) = 1.0 then

  update := 1;
else
  update := 0;

end if;

return update;

end correct;


function compute_ratio ( numerator   : float;
                   denominator : integer ) return float is

quotient : float;

begin

quotient := numerator / float(denominator);

return quotient;

end compute_ratio;


function compute_average ( perf_array : matrix;
                     tot_passes : integer ) return vector is

temp        : float;
perf_vector : vector ( perf_array'range(2) );

begin

for j in perf_array'range(2) loop
  temp := 0.0;
  for i in perf_array'range(1) loop
    temp := temp + perf_array(i, j);
  end loop;
  perf_vector(j) := temp / float(tot_passes);
end loop;
```

```
return perf_vector;

end compute_average;

procedure initialize_networ: ( L1, L2, L3 : in out layer;
        center      : in float;
        width       : in float;
        seed        : in out system.unsigned_longword ) is

begin

for j in L1.W'range(2) loop
  for i in L1.W'range(1) loop
    uniform ( center, width, seed, L1.W(i, j) );
    L1.Del_W(i, j) := 0.0;
  end loop;
  uniform ( center, width, seed, L1.Theta(j) );
  L1.Del_Theta(j) := 0.0;
end loop;

for i in L1.Fin'range loop
  L1.Fin_Prime(i) := 0.0;
end loop;

for j in L2.W'range(2) loop
  for i in L2.W'range(1) loop
    uniform ( center, width, seed, L2.W(i, j) );
    L2.Del_W(i, j) := 0.0;
  end loop;
  uniform ( center, width, seed, L2.Theta(j) );
  L2.Del_Theta(j) := 0.0;
end loop;

for j in L3.W'range(2) loop
  for i in L3.W'range(1) loop
    uniform ( center, width, seed, L3.W(i, j) );
    L3.Del_W(i, j) := 0.0;
  end loop;
  uniform ( center, width, seed, L3.Theta(j) );
  L3.Del_Theta(j) := 0.0;
end loop;

end initialize_network;


procedure forward_pass ( L      : in out layer;
                         A3, A5 : in float ) is

Temp : float;

begin
```

```
L.X    := L.Fin * L.W;
L.Fout := sigmoid ( L.X + L.Theta );
L.V    := ( others => 0.0 );

for j in L.W'range(2) loop
  for i in L.W'range(1) loop
    L.V(j) := L.V(j) + ( L.Fin(i) * ( A3 * L.W(i, j)
                         + A5 * L.Del_W(i, j) ) )
                         + ( L.Fin_Prime(i) * L.W(i, j) );
  end loop;

  Temp            := ( A3 * L.Theta(j) )
                       + ( A5 * L.Del_Theta(j) );

  L.V(j)          := L.V(j) + Temp;
  L.U(j)          := L.Fout(j) * ( 1.0 - L.Fout(j) );
  L.Fout_Prime(j) := L.V(j) * L.U(j);
end loop;

end forward_pass;


procedure compute_forward_pass ( L1, L2, L3 : in out layer;
                                 A3, A5     : in float ) is

begin

forward_pass ( L1, A3, A5 );

L2.Fin       := L1.Fout;
L2.Fin_Prime := L1.Fout_Prime;

forward_pass ( L2, A3, A5 );

L3.Fin       := L2.Fout;
L3.Fin_Prime := L2.Fout_Prime;

forward_pass ( L3, A3, A5 );

end compute_forward_pass;


procedure backward_pass ( L      : in out layer;
                          A3, A5 : in float ) is

begin

for j in L.W'range(2) loop
  L.Q(j)  := L.U(j) * L.Etotal(j);

  L.R(j)  := L.U(j) * ( L.Etotal_Prime(j) + ( L.Etotal(j)
                    * ( 1.0 - 2.0 * L.Fout(j) ) * L.V(j) ) );
  for i in L.W'range(1) loop
    L.Eout(i, j)       := L.Q(j) * L.W(i, j);
    L.Eout_Prime(i, j) := ( L.R(j) * L.W(i, j) )
                          + ( L.Q(j) * ( A3 * L.W(i, j)
                          + A5 * L.Del_W(i, j) ) );
  end loop;
end loop;


end backward_pass;


procedure update_weights ( L          : in out layer;
                           cost       : in out float;
                           A1, A2, A4 : in float ) is

begin

for j in L.W'range(2) loop
  for i in L.W'range(1) loop
    L.Del_W(i, j) := ( ( 1.0 - A4 ) * L.Del_W (i, j) )
                 - ( A2 * L.W(i, j) )
                 + ( ( ( A1 * L.Q(j) ) + L.R(j) ) * L.Fin(i) )
                 + ( L.Q(j) * L.Fin_prime (i) );

    L.W(i,j)      := L.W(i,j) + L.Del_W(i, j);

    cost := cost + ( L.Del_W(i, j) ** 2 );

  end loop;
end loop;

end update_weights;


procedure update_thresholds ( L          : in out layer;
                              cost       : in cut float;
                              A1, A2, A4 : in float ) is

begin

for i in L.Theta'range loop
  L.Del_Theta(i) := ( ( 1.0 - A4 ) * L.Del_Theta(i) )
                    - ( A2 * L.Theta(i) )
                    + ( ( A1 * L.Q(i) ) + L.R(i) );

  L.Theta(i)     := L.Theta(i) + L.Del_Theta(i);

  cost := cost + ( L.Theta(i) ** 2 );
end loop;
end update_thresholds;


procedure find_max_vals ( output : in vector;
                          index1 : in out integer;
                          index2 : in out integer ) is

max1  : float := 0.0;
max2  : float := 0.0;
temp1 : integer;
temp2 : integer;

begin

for i in output'range loop
  if output(i) >= max1 then
    temp1 := i;
    max1  := output(i);
  end if;
end loop;

for i in output'range loop
  if output(i) >= max2 and output(i) < max1 then
    temp2 := i;
    max2  := output(i);
  end if;
end loop;

index1 := temp1;
index2 := temp2;

end find_max_vals;


end somp_support;
```

```
with system;
with text_io;                use text_io;
with float_text_io;          use float_text_io;
with integer_text_io;        use integer_text_io;

with vector_operations;      use vector_operations;
with math_lib_extension;     use math_lib_extension;

package somp_io is

procedure get_moment_array ( image_list   : in string;
                             num_moments  : in integer;
                             moment_array : in out matrix );

procedure get_feature_arrays ( num_features  : in integer;
                               filename      : in out string;
                               feature_array : in out matrix );

procedure generate_random_moms ( num_patterns : in integer;
                      num_moments  : in integer;
                      moment_array : in matrix;
                      input        : in out vector;
                      Dout         : in out vector;
                      seed : in out system.unsigned_longword );

procedure generate_seq_moms ( count        : in integer;
                              num_moments  : in integer;
                              moment_array : in matrix;
                              input        : in out vector;
                              Dout         : in out vector );

procedure store_net_perf ( perf_vector : in vector;
                       perf_file   : in out text_io.file_type;
                       interval    : in integer );

end somp_io;
```

```
package body somp_io is

procedure get_moment_array ( image_list   : in string;
                       num_moments  : in integer;
                       moment_array : in out matrix ) is

initial    : integer := 1;
final      : integer := 0;
temp_class : integer;
num_vector : integer;

image_name : string ( 1 .. 14 );
image_file : text_io.file_type;
Temp_name  : text_io.file_type;

begin

open ( image_file, in_file, image_list );

while not end_of_file ( image_file ) loop
  get ( image_file, image_name );
  open ( temp_name, in_file, image_name );
  get ( temp_name, temp_class );
  get ( temp_name, num_vector );

  final := final + num_vector;

  for i in initial .. final loop
    skip_line ( temp_name, 3 );
    for j in 2 .. ( num_moments + 1 ) loop
      moment_array(i, 1) := float(temp_class);
      get ( temp_name, moment_array(i, j) );
    end loop;
  end loop;

  close ( temp_name );

  initial := initial + num_vector;

end loop;

close ( image_file );

end get_moment_array;


procedure get_feature_arrays ( num_features  : in integer;
                         filename      : in out string;
                         feature_array : in out matrix ) is

counter    : integer := 1;
temp_class : integer;

target     : string ( 1 .. 4 );
```

```
input_file : text_io.file_type;

begin

open ( input_file, in_file, filename );
while not end_of_file (input_file) loop

  get ( input_file, temp_class );
  get ( input_file, target );
  skip_line ( input_file );

  feature_array(counter, 1) := float( temp_class );

  for j in 2 ..( num_features + 1 ) loop
    get ( input_file, feature_array(counter, j) );
  end loop;

  skip_line ( input_file, 2 );
  counter := counter + 1;

end loop;

close ( input_file );

end get_feature_arrays;


procedure generate_random_moms ( num_patterns : in integer;
                     num_moments  : in integer;
                     moment_array : in matrix;
                     input        : in out vector;
                     Dout         : in out vector;
                     seed : in out system.unsigned_longword ) is

pick       : float;
temp_class : integer;
choice     : integer;

begin

Dout := ( others => 0.0 );

uniform ( 0.5, 0.5, seed, pick );
choice := integer ( pick * float(num_patterns) + 0.5 );
if choice < 1 then
  choice := 1;
elsif choice > num_patterns then
  choice := num_patterns;
end if;
```

```
temp_class := integer ( moment_array(choice, 1) );

for j in 2 .. ( num_moments + 1 ) loop
  input(j - 1) := moment_array(choice, j);
end loop;

Dout(temp_class) := 1.0;

end generate_random_moms;


procedure generate_seq_moms ( count        : in integer;
                        num_momerts  : in integer;
                        moment_array : in matrix;
                        input        : in out vector;
                        Dout         : in out vector ) is

temp_class : integer;

begin

Dout       := ( others => 0.0 );
temp_class := integer(moment_array(count, 1));

for j in 2 .. ( num_moments + 1 ) loop
  input(j - 1) := moment_array(count, j);
end loop;

Dout(temp_class) := 1.0;

end generate_seq_moms;


procedure store_net_perf ( perf_vector : in vector;
                     perf_file   : in out text_io.file_type;
                     interval    : in integer ) is

temp_interval : integer := 0;

begin

put ( perf_file, "x = [" );
new_line ( perf_file );

for i in perf_vector'range loop
  put ( perf_file, temp_interval );
  new_line ( perf_file );
  temp_interval := temp_interval + interval;
end loop;

put ( perf_file, "]" );
new_line ( perf_file );
put ( perf_file, "y = [" );
new_line ( perf_file );

for i in perf_vector'range loop
  put ( perf_file, perf_vector(i) );
  new_line ( perf_file );
end loop;

put ( perf_file, "]" );

end store_net_perf;


end somp_io;
```

```
--***********************************************************
--
--             Math_Lib_Extension
--
--***********************************************************
--
--  Purpose: Provides access to pseudorandom number
--  generators for both uniform and Gaussian distributions.
--
--  Inputs:   1) See individual routines.
--
--  Outputs:  1) See individual routines.
--
--  Author: Dennis W. Ruck (GE-87D), AFIT/ENG
--
--
--***********************************************************

with system;                  use system;
package math_lib_extension is

type time_array is new unsigned_word_array (1..7);

  procedure mth_random ( val : out float; seed : in out unsigned_longword );
  pragma INTERFACE ( vaxrtl, mth_random );
  pragma IMPORT_VALUED_PROCEDURE ( mth_random, "MTH$RANDOM",
                                   mechanism => (value, reference));

  procedure uniform ( center  : in float;
                      width   : in float;
                      seed    : in out unsigned_longword;
                      val     : out float );

  procedure gaussian ( mean      : in float;
                       variance  : in float;
                       seed      : in out unsigned_longword;
                       val       : out float );

  function get_seed return unsigned_longword;

end math_lib_extension;
```

```
--***********************************************************
--
--             Math_Lib_Extension (package body)
--
--***********************************************************
--
--  Purpose: Package to provide pseudeorandom number generators
--  with uniform and Gaussian distributions.
--
--  Inputs:   1) See the individual routines.
--
--  Outputs:  1) See the individual routines.
--
--  Author: Dennis W. Ruck (GE-87D), AFIT/ENG
--
--
--***********************************************************

with starlet;
with condition_handling;
with float_math_lib;            use float_math_lib;

package body math_lib_extension is

-- This procedure will return a uniform random sample centered
-- about CENTER plus or minus WIDTH.
procedure uniform ( center  : in float;
                    width   : in float;
                    seed    : in out unsigned_longword;
                    val     : out float ) is
x      : float;

begin

  -- get uniform random variable between 0 and 1
  mth_random ( x, seed );

  -- adjust to center +/- width
  x := x * width * 2.0;
  x := x + center - width;

  val := x;

end uniform;
```

```
-- This procedure will return a gaussian random variable sample
-- with mean MEAN  and variance of VARIANCE. The central limit theorem
-- is invoked to approximate the gaussian with a sum of uniform RVs.
procedure gaussian ( mean      : in float;
                     variance  : in float;
                     seed      : in out unsigned_longword;
                     val       : out float ) is

num_rvs              : constant := 20;
sum                  : float := 0.0;
x                    : float;
Z                    : float;
Y                    : float;
ave                  : float;
norm                 : float;

begin

  -- Obtain a sum of random variables that are uniform between
  -- 0 and 1.
  for i in 1..num_rvs loop
    mth_random ( x, seed );
    sum := sum + x;
  end loop;

  ave := sum / float(num_rvs);

  -- AVE is a rv with mean = 0.5 and variance = 1/(12*num_rvs);
  -- now normalize AVE
  Z := (ave-0.5)/sqrt(1.0/(12.0*float(num_rvs)));

  -- Now unnormalize to desired mean and variance
  Y := mean + sqrt(variance)*Z;

  val := Y;

end gaussian;

function get_seed return unsigned_longword is
--
-- Returns the lower unsigned longword of the binary representation
-- of the system time as the initial seed for the pseudo-
-- random number generator.
```

```
--
status      : condition_handling.cond_value_type;
bintim      : unsigned_quadword;

begin

  STARLET.gettim ( status, bintim );

  return bintim.LO;

end get_seed;

end math_lib_extension;
```

```
--************************************************************
--
--             Vector_Operations (package spec)
--
--************************************************************
--
--  Purpose: Provide general vector operations to allow
--  a more readable implementation of equations consisting
--  of one and two dimensional arrays.
--
--  Inputs:   1) See individual routines.
--
--  Outputs:  1) See individual routines.
--
--  Author: Dennis W. Ruck (GE-87D), AFIT/ENG
--  Modified By: Charles C. Piazza (GE-88D), AFIT/ENG
--
--************************************************************

with text_io;
package vector_operations is

type vector is array ( integer range <> ) of FLOAT;

type matrix is array ( integer range <>, integer range <> ) of FLOAT;

function "*" ( left   : vector;
               right  : matrix ) return vector;

function "*" ( left, right : vector ) return float;

function "*" ( left  : float;
               right : matrix ) return matrix;

function "*" ( left  : float;
               right : vector ) return vector;

function "+" ( left, right  : matrix ) return matrix;

function "+" ( left, right : vector ) return vector;

function "-" ( left, right : matrix ) return matrix;

function "-" ( left, right : vector ) return vector;
function distance ( left, right : vector ) return float;

procedure put ( output : in text_io.file_type;
                data   : in vector );

procedure put ( data   : in matrix );

procedure get ( input  : in text_io.file_type;
                data   : out vector );

end vector_operations;
```

```
--***********************************************************
--
--             Vector_Operations (package body)
--
--***********************************************************
--
--  Purpose: Provide generic vector operations to allow a more
--  readable implementation of equations consisting of one and
--  two dimensional arrays.
--
--  Inputs:   1) See individual routines.
--
--  Outputs:  1) See individual routines.
--
--  Author: Dennis W. Ruck (GE-87D), AFIT/ENG
--  Modified By: Charles C. Piazza (GE-88D), AFIT/ENG
--
--***********************************************************

with text_io;                   use text_io;
with float_text_io;             use float_text_io;
with float_math_lib;            use float_math_lib;

package body vector_operations is

function "*" ( left   : vector;
               right  : matrix ) return vector is

sum      : FLOAT;
product  : vector ( right'range(2) );

begin

  for j in right'range(2) loop
    sum := 0.0;
    for i in right'range(1) loop
      sum := sum + left (i) * right (i,j);
    end loop;
    product (j) := sum;
  end loop;

  return product;

end "*";
```

```
function "*" ( left, right : vector ) return FLOAT is

sum : FLOAT := 0.0;

begin

  for i in left'range loop
    sum := sum + left (i) * right (i);
  end loop;

  return sum;

end "*";


function "*" ( left  : float;
               right : matrix ) return matrix is

product : matrix ( right'range(1), right'range(2) );

begin

  for i in right'range(1) loop
    for j in right'range(2) loop
      product (i, j) := left * right (i, j);
    end loop;
  end loop;

  return product;

end "*";


function "*" ( left  : float;
               right : vector ) return vector is

product : vector ( right'range );

begin

for i in right'range loop
  product (i) := left * right (i);
end loop;
```

```
return product;

end "*";


function "+" ( left, right : matrix ) return matrix is

sum : matrix ( left'range(1), left'range(2) );

begin

for i in left'range(2) loop
  for j in left'range (1) loop
    sum (i, j) := left (i, j) + right (i, j);
  end loop;
end loop;

return sum;

end "+";


function "+" ( left, right : vector ) return vector is

sum : vector ( left'range );

begin

  for i in left'range loop
    sum (i) := left (i) + right (i);
  end loop;

  return sum;

end "+";


function "-" ( left, right : matrix ) return matrix is

diff : matrix ( left'range(1), left'range(2) );

begin

for i in left'range(2) loop
  for j in left'range(1) loop
```

```
    diff (i, j) := left (i, j) - right (i, j);
  end loop;
end loop;

return diff;

end "-";


function "-" ( left, right : vector ) return vector is

diff : vector ( left'range );

begin

  for i in left'range loop
    diff (i) := left (i) - right (i);
  end loop;

  return diff;

end "-";


function distance ( left, right : vector ) return float is

sum_x2    : float := 0.0;

begin

  for j in left'range loop
    sum_x2 := sum_x2 + ( left (j) - right (j) ) * ( left (j) - right (j) );
  end loop;

  return sqrt ( sum_x2 );

end distance;


procedure put ( output : in text_io.file_type;
                data   : in vector ) is

col_max  : constant := 72;
width    : constant := 10;
col      : positive := 1;
```

```
begin
  for j in data'range loop
    put ( output, data (j), 0, 6, 1 );
    put ( output, " " );
    col := col + width;
    if col > col_max then
      new_line ( output );
      col := 1;
    end if;
  end loop;
end put;


procedure put ( data   : in matrix ) is

col_max  : constant := 72;
width    : constant := 10;
col      : positive := 1;

begin
  for i in data'range(1) loop
    for j in data'range(2) loop
      put ( data (i,j), 1, 4, 0 );
      put ( " " );
      col := col + width;
      if col > col_max then
        new_line;
        col := 1;
      end if;
    end loop;
    new_line;
    col := 1;
  end loop;
end put;


procedure get ( input : in  text_io.file_type;
                data  : out vector ) is
begin
  for j in data'range loop
    get ( input, data (j) );
  end loop;
end get;
end vector_operations;
```

Bibliography

[1] Bendat, Julius Bendat and Piersol, Allan G. "Random Data: Analysis and Measurement Procedures", Wiley-Interscience, 99-105.

[2] Hu, Ming-Kuei, "Visual Pattern Recognition by Moment Invariant", IRE Trans. Inform. Theory, vol. IT-8 179-187 (February 1962).

[3] Kreyszig, Erwin, "Advanced Engineering Mathematics". Fifth Edition, John Wiley & Sons, 764.

[4] Lippmann, Richard P. "An Introduction to Computing with Artificial Neural Networks", IEEE Transactions on Computers, 4:4-22 (April 1987).

[5] Melsa, James L. and Cohn, David L. "Decision and Estimation Theory", McGraw-Hill Book Company, 1-53.

[6] Oxley, Mark, Asst. Professor. Notes taken during interviews and discussions. Department of Mathematics and Computer Science, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, 1988.

[7] Parker, David B. "Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning", Proceedings of the IEEE First Annual Conference on Neural Networks, vol. II, 593-600 (June 1987).

[8] Parker, David B. "Second Order Back Propagation: Implementing an Optimal O(n) Approximation to Newton's Method as an Artificial Neural Network", for submission to Computer, (September 1987).

[9] Rogers, Steven K. Asst. Professor. Notes taken from weekly meetings. Department of Electrical Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, 1988.

[10] Roggemann, Mike. Phd Student. Notes taken from interviews and discussions. Provided data for testing and analysis. Department of Electrical Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, 1988.

[11] Rosenfeld, Azriel. "Digital Picture Processing", vol. II, second edition, 62-73.

[12] Ruck, Dennis W. "Multisensor Target Detection and Classification", MS Thesis, AFIT/GE/87D-56. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, (December 1987).

[13] Rummelhart, David E., Hinton, Geoffrey E., and Williams, Ronald J. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1:Foundations. Cambridge MA: MIT Press, 1986.

[14] Strang, Gilbert. "Linear Algebra and Its Applications", second edition, Academic Press, 243-254; 297-304.

[15] Werbos, Paul J. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences", Harvard University (August 1974).

[16] Williams, Ronald J. "On the Use of Backpropagation in Associative Reinforcement Learning", Proceedings of the IEEE Second Annual Conference on Neural Networks, vol. I, 263-273 (June 1988). (June 1987).

VITA

Captain Charles C. Piazza was born on [redacted] He graduated from high school in [redacted] in 1976. He enlisted into the United States Air Force in 1977 and was accepted into the Airman Education and Commissioning Program in March 1981. Captain Piazza attended the University of South Carolina and received the degree of Bachelor of Science in Computer and Electrical Engineering in May 1984. Upon completion of Officer Training School, he received his commission in September 1984. Captain Piazza was assigned to the 1815th Operational, Test and Evaluation Squadron, in January 1985. While there, he performed the duties of Team Chief, of a highly technical high frequency evaluation team. He was also selected as the Course Director of the Narrowband High Frequency Systems Evaluation Course, for AFCC's Systems Evaluation School in March 1986. In May of 1987, he entered into the School of Engineering, Air Force Institute of Technology.

[redacted]

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
AFIT/GE/ENG/88D-36

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
School of Engineering

6b. OFFICE SYMBOL *(If applicable)*
AFIT/ENG

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS *(City, State, and ZIP Code)*
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

7b. ADDRESS *(City, State, and ZIP Code)*
12 Jan 1989

8a. NAME OF FUNDING / SPONSORING ORGANIZATION RADC
Fred Diamond

8b. OFFICE SYMBOL *(If applicable)*
COTC

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
Griffis AFB NY 13411

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Modified Backward Error Propagation for Tactical Target Recognition

12. PERSONAL AUTHOR(S)
Charles C. Piazza, B.S., Capt, USAF

13a. TYPE OF REPORT
Thesis

13b. TIME COVERED
FROM ________ TO ______

14. DATE OF REPORT *(Year, Month, Day)*
1988 December

15. PAGE COUNT
198

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	09	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Artificial Neural Network, Backward Error Propagation, Pattern Classification, Target Recognition, Learning Machines

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This thesis explores a new approach to the classification of tactical targets using a new biologically-based neural network. The targets of interest were generated from doppler imagery and forward looking infrared imagery, and consisting of tanks, trucks, armored personnel carriers, jeeps and petroleum, oil, and lubricant tankers. Each target was described by feature vectors, such as normalized moment invariants. The features were generated from the imagery using a segmenting process. These feature vectors were used as the input to a neural network classifier for tactical target recognition.

The neural network consisted of a multilayer perceptron architecture, employing a backward error propagation learning algorithm. The minimization technique used was an approximation to Newton's method. This second order algorithm is a generalized version of well known first order techniques, i.e., gradient of steepest descent and momentum methods. Classification using both first and second order techniques was performed, with comparisons drawn.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
□ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Steven Rogers, Captain, USAF

22b. TELEPHONE *(Include Area Code)*
(513) 255-3576

22c. OFFICE SYMBOL
AFIT/ENG

Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE