A CONJUGATE-GRADIENT BASED ALGORITHM TO TRAIN
FEEDFORWARD MULTILAYER PERCEPTRONS
A DISSERTATION

by

RAMONA ROSARIO

Approved as to style and content by:

N. Tepedelenlioglu, Ph. D., Committee Chairman
Associate Professor of Electrical and Computer Engineering

C.D. Beach, Ph. D., Member
Associate Professor of Electrical and Computer Engineering

D.W. Fausett, Ph. D., Member
Professor of Applied Mathematics

F. Ham, Ph. D., Member
Associate Professor of Electrical and Computer Engineering

R. L. W. Welch, Ph. D., Member
Adjunct Professor of Mathematical Sciences

T.J. Sanders, Ph.D., Department Head
Professor of Electrical and Computer Engineering
ABSTRACT

An algorithm to train feedforward multilayer perceptrons is presented. This algorithm trains the perceptron in fewer pattern presentations than standard backpropagation. A parallel processing scheme renders it computationally efficient.

The algorithm uses the error backpropagated from the output layer to each node in the network to formulate a normal equation at each node. These equations are solved at every node using the method of conjugate-gradient directions.

For the vector classification problem considered (circle in the square), this algorithm trains 20 times faster than backprop. For the pattern recognition type problems considered, the implementation in parallel architecture of this algorithm would train in half the time of the implementation in parallel architecture of standard backpropagation.

Such an algorithm would be useful in time-critical applications where there is a steady stream of training data at a fixed rate. It would train using far less information than backpropagation.
To my genius friend,

Francis Michael Rotella,

without whom it might very well have remained

only a thought
ACKNOWLEDGEMENTS

I owe gratitude to many. I hope I can name them all here.

Above all I thank Dr. Nazif Tepedelenlioglu whose good humour, wit and patience made this endeavour as close to a pleasure as anything of this kind can be. He was my muse who rescued me from despair and ignorance often, who read and re-read this "tome" endless times and helped me find the confidence to forge on.

I thank my parents Rose and Dominic Rosario who have always had faith in me throughout the years.

I owe gratitude to Ali Rezgui who helped me program the first algorithms I attempted. Bob Scalerio offered me invaluable advice during my qualifying exams and I have found his work on the recursive least squares based algorithm indispensable.

I thank all my other committee members Dr. Charles D. Beach, Dr. Frederic M. Ham, Dr. Donald W. Fausett and Dr. Welch for their advice and guidance which made this a better work.

I also wish to thank Dr. T. J. Harrison at the FAMU / FSU College of Engineering, Tallahassee, FL. for providing me with the resources and time to complete the writing. I thank David Haff for helping me with the figures and then driving at breakneck speed to catch the FEDEX office in time. I thank Mary Jo Schmidt for cheerfully doing the extra typing.
Of all who helped, none took more pains than Francis M. Rotella who stayed up all night with me the day before my proposal, who taught me how to use the SUNs and who wrote the scripts which enabled me to run the experiments.

I express my gratitude to Dr. Fatih Unal for his advice on parallel processing.

Other people who I bugged for help at various other times and who sweetly obliged include Mark D'souza, Bill Czagas and Joe Smith. Thank-you guys!

And last but not the least I wish to thank all the lovely people who have been my friends and buoyed me up emotionally: Ramzi Nassar, John Bard, Nora King, Dr. T, Eileen Kennelly, Zachary Ashton, Farzana McRae, Semen Koksal, Nina Adam Glenn, Sevim Topcu, Malvina Rosado Hall, Michele Raymond, Steven Hathaway, Pierre Leger, Isabelle and Herve Schwertz, Gloria Beckwith, Stella Stehno, Stephanie Cioffi Franke, Vickie, Sam Kozaitis, Zahia Drici, and Gian Marco Cabibbe. Thanks for putting up with me.
# TABLE OF CONTENTS

## ONE

**Introduction**  
1.1 Parallel Distributed Processing: A New Paradigm  
1.2 Learning in a Neural Network  
1.3 Purpose  

## TWO

**The Normal Equation and the Supervised Training of a Multilayer Feedforward Perceptron**  
2.1 The Multilayer Feedforward Perceptron and the Linear Combiner  
2.2 The Nonnal Equation and the Linear Combiner  
2.3 The LMS Algorithm  
2.4 Backprop  
2.5 A Generalized Approach to the Training of the Feedforward Multilayer Perceptron  

## THREE

**Training Algorithm with Normal Equation Solved by the Method of Conjugate Gradients**  
3.1 The Method of Conjugate Gradients to solve the Normal Equation  
3.2 The Training Algorithm  
3.3 Other Conjugate Gradient Training Algorithms  

## FOUR

**Experimental Results**  
4.1 Circle in the Square  
4.2 Weakly Correlated Patterns  
4.3 Strongly Correlated Patterns  
4.4 Computational Considerations  

## FIVE

**A Parallel Processing Scheme to Implement the Conjugate-Gradient Based Algorithm**  
5.1 Parallelizing the Conjugate-Gradient Update  
5.2 The Parallel Processing Scheme  

## SIX

**Conclusions**  

**Appendix**  
Derivation of Parallel Form of the Conjugate-Gradient Update  

**References**
FIGURES

1.1 A Multilayer Feedforward Perceptron 8
1.2 A Node: Main Component of a Multilayer Feedforward Perceptron 9
2.1 A Linear Combiner 18
2.2 Diagram of a Multilayer Feedforward Perceptron Illustrating Notation Used 24
4.1 Performance of the Three Algorithms for the Circle in the Square Problem 48
4.2 Weakly Correlated Patterns 50
4.3 Mean-Squared Error vs. Iteration for Weakly Correlated Patterns 51
4.4 Strongly Correlated Patterns 54
4.5 Mean-Squared Error vs. Iteration for Strongly Correlated Patterns 55
5.1 Computation of the Conjugate-Gradient Weight Update 63
5.2 Data Flow Among the Processors During Computation of the Conjugate-Gradient Weight Update 64
5.3 Computation of the Backprop Weight Update 65
CHAPTER 1: INTRODUCTION

1.1: Parallel Distributed Processing: A New Paradigm

Ever since the forties [1], efforts have been underway to make machines capable of simulating human cognition. The established models based on Von Neumann serial processing for describing mental processes did not adequately approximate human perception, memory and thought.

Cognition is the activity of knowing, the acquisition, organization and use of knowledge. Humans are continually involved in the activity of knowledge. We make decisions, discern contextual relationships and recognize patterns based on incomplete and even erroneously specified data. By contrast, the performance of sequential information processing hinges on the accuracy and precision of available data. Algorithms based on the rigid adherence to a stringently specified sequence of steps are inherently incapable of the simultaneous resolution of several relevant factors necessary to conduct cognitive tasks such as pattern recognition and associative memory recall. The current consensus is that such tasks are best accomplished by an essentially different class of models -- Parallel Distributed Processing (PDP) models [2]. These models are based on the premise that cognition is effected by a structure which distributes processing among a large number of simple interconnecting computational units. Such processing considers several possible solutions simultaneously. Another popular term for PDP models is "Neural Networks" alluding to their origin in attempts to mimic the behavior of the human brain.
Various PDP models have been devised [3] such as Hopfield nets [4], perceptrons [5], Hamming nets, ART (Adaptive Resonance Theory) devised by Carpenter and Grossberg [6, 7], and Kohonen nets [8]. The differences between the models are significant. Yet they share the common feature that without exception they are all composed of several simple, identical computational units. The response of a neural network to stimuli (applied inputs, i.e., data) is governed not by a program but by the interactions among the computational units.

What procedure determines the interaction structure which solves the relevant problem? The metaphor to the human brain is not restricted to their name: neural networks learn the correct interactions. The manner in which this learning occurs is one of the many ways in which a particular PDP model can differ from another. The types of learning mechanisms will be discussed in the next section.
1.2: Learning in a Neural Network.

A neural network undergoes a training phase during which evolves the required interaction structure between its computational units or nodes. A correct interaction structure is one which enables the net to perform the information processing task required of it.

Nodes in a neural network are connected to some or all other nodes depending on the type of network. The connections between the nodes are manifested in the following manner: a node's output is multiplied by a real-valued number before it contributes to the input of another node to which it is connected. Since a node's output is "weighted" by this number, these connections are termed "weights".

Weights thus govern the interactions between nodes. A learning scheme brings these weights from usually initial random values to those values which will elicit the correct response from the network to applied stimuli.

A learning model is an incorporation of rules which dictate the method of presenting data to the network and the manner in which the weights should change according to the response of the network. Learning models can be classified along a spectrum, at one end of which is learning with an error-correcting teacher (supervised learning) and at the other is spontaneous, unsupervised discovery (unsupervised learning). Between these two extremes lies what is sometimes called "forced learning" which consists of rules to
manipulate the content of the input data to bring about learning [2].

Despite the fact that learning mechanisms lie along a spectrum, a useful classification based on four common learning paradigms in which some PDP models apparently work can be made [2]:

Auto Associator: the training phase consists of repeatedly presenting patterns to the network which changes its weight structure in order to "store" the patterns. After training is completed, the network retrieves the original pattern when presented with a part or a degraded version of the original pattern as an input [2]. One of the Hopfield models falls into this category. Hopfield nets are normally used with binary inputs. They can be used as an associative memory or to solve optimization problems. When used as an associative memory, the weights are chosen during the training phase according to a recipe dictated by the training patterns (exemplars). Upon completion of training, an input pattern is presented by forcing the output of the nodes to coincide with it. Every node in a Hopfield network is connected to another node. Hence the output changes until it stabilizes. Stabilization is ensured by imposing certain symmetry conditions on the weights. The outputs change until they coincide with the exemplar to which the input pattern belongs or until a no match condition is signalled [9].

Pattern Associator: the training phase consists of repeatedly presenting pairs of patterns to the network. After training is completed, the network is expected to respond to a member of the pair applied as the input by presenting the other member of the pair, thus
mapping a set of input patterns to a set of output patterns [2].

Classification Paradigm: the training phase consists of repeatedly presenting patterns along with the categories to which each pattern belongs. After training is complete the net is expected to categorize a pattern or a slightly distorted version of the pattern correctly.

Perceptrons are designed to operate according to this paradigm. Another example of such training is the Hamming net which is trained so that only one output node signifying the category of the applied input pattern comes "on" [2].

These paradigms are examples of supervised learning. The training phase includes the provision of the desired response. The next paradigm models unsupervised learning since the desired response is not provided during training.

Regularity Detector: the training phase consists of presenting patterns from a population of patterns with a certain probability. The network is expected to discover statistically salient features of the input pattern set even though there is no specification of categories into which the patterns must be classified. Networks trained under this paradigm are Kohonen self-organizing nets and Carpenter/Grossberg nets. They perform functions such as data clustering and feature detection [2].

In summary, learning in neural networks falls broadly into two categories: Supervised and Unsupervised. Supervised learning is that learning which occurs in the presence of a
"teacher" which provides the correct response. Unsupervised learning does not employ knowledge of a correct response. An unsupervised learning rule updates the weights of the network when input data are applied in such a way that the network's response reflects hidden correlations in the data thus creating categories based on the correlations between the input data.

No learning model, however, can train a net whose architecture is inadequate to perform the task expected of it. Whether a neural network learns an appropriate task or not is completely dependent on the number of nodes and the kind of computation each node performs. Rumelhart et al. make this point very clear:

... no network can learn what it is not capable of doing in principle. What any particular network can do is dependent on its structure and the computational properties of its component elements. Unfortunately there is no canonical way to find the best network or to determine what it will learn, so the whole enterprise still has much of the flavor of an experimental science [9].

Research effort [10,11] is directed towards determining the adequate architecture of a net for a particular task.
1.3: Purpose

This dissertation focuses on the supervised training of the multilayer feedforward perceptron. The multilayer feedforward perceptron is a neural network which has found wide and varied application in many fields. It has been used successfully for image and speech processing, in character recognition systems, pattern recognition, loan application processing, controllers, nuclear plant safety monitoring, and image compression.

Figure 1.1 shows a diagram of a multilayer feedforward perceptron. The circles represent nodes which each comprise a linear combiner and a nonlinear function. The linear combiner generates the inner product of the input vector to the node and the weight vector associated with the node. The nonlinear function acts upon this inner product to generate the output of the node. Figure 1.2 shows a representation of a single node. The arrows represent the weights connecting nodes to each other. The term multilayer refers to the presence of one or more hidden layers of nodes between the input and output layer. In the feedforward architecture, the nodes are arranged in consecutive layers, with inputs to nodes in the first layer being the inputs to the net, and outputs of nodes in the last layer, being the output of the net. Inputs to nodes in the other layers, the hidden layers, are the outputs of nodes in the immediately preceding layer. The output of nodes in any layer other than the outer layer comprise the input to nodes in the immediately consecutive layer [12].
Figure 1.1 A Multilayer Feedforward Perceptron
Figure 1.2 A Node: Main Component of a Multilayer Feedforward Perceptron
The multilayer feedforward perceptron is an example of a network trained under the classification paradigm. Once trained, it classifies patterns into specified categories. Since there are no constraints on the specified categories, the multilayer feedforward perceptron can be trained to perform as a pattern associator and also as an auto-associator yielding the original pattern when presented with a portion or a distorted version of the pattern. In fact, in the image compression scheme devised by Cottrell/Munro and Zipser [13], the clever ploy of using the image to be compressed as both input pattern and desired response, and having a smaller number of nodes in the hidden layer causes the network to summarize the essential features of the image to be compressed at the hidden layer, thus acting as a feature detector.

Supervised training of the multilayer feedforward perceptron to perform a desired mapping is carried out in the following fashion: input vectors chosen from a training set are applied in turn to the net and the actual output vector of the net is compared to the desired output (the vector to which the network is expected to map the input vector). The difference between the actual and desired output, i.e. the error, is used to update the weights. This procedure is repeated until the squared error averaged over a test set is below a specified tolerance. Several input vectors can also be mapped to a single output vector.

The availability of the error at each node is essential to the supervised training of the perceptron. The desired output of every node of the single-layer perceptron is explicitly specified as every node of a single-layer perceptron is an output node. This is why a
single-layer perceptron is considered more trainable than the multilayer perceptron. In the case of the multilayer perceptron, however, one has the problem that the desired output of the nodes in the hidden layers is unknown.

In spite of this obstacle to their training, multilayer perceptrons come in useful when a particular application calls for a non-linearly separable mapping [3]. Unlike single-layer perceptrons, multilayer perceptrons are not limited in performance to the so-called linearly separable problems.

It is the addition of hidden layers, however, which hinders the specification of a correct response for nodes in the hidden layers. Until recently, this prevented the development of effective supervised training algorithms which resulted in limited application for multilayer nets. The development which makes the training of a multilayer feedforward perceptron possible is the backpropagation algorithm (backprop for short) [2, 3, 14-17].

Backprop is an iterative, gradient minimization algorithm. The backprop weight update consists of a small negative fraction of the derivative of the mean squared error at the output of the net with respect to the weight to be updated [2].

Backprop is a simply formulated and easily implemented algorithm, yet in many cases it does not meet the desired time constraints. Commercial applications of the multilayer perceptron demand speed in training. Sometimes the number of presentations of training data required for the mean squared error to fall to an acceptable level has to be large [2]
and training can even fail when the network gets stuck in a local minimum. Hence considerable research is directed towards devising faster and more reliable alternatives to backprop.

This dissertation utilizes a useful insight for the formulation of a whole class of fast algorithms. Despite the relative novelty of PDP models, this insight enables well-established adaptive filtering techniques to bear on the problem of training the feedforward multilayer perceptron.

The insight used is the realization that classical backprop can be interpreted as carrying out two functions: backpropagating the error from the nodes in the outer layer to nodes in the hidden layer and using this error in an LMS update of the weights [9].

The realization that these two functions -- estimation of error and updating of weights -- are distinct and separate has an important ramification: it allows for variations on the weight update mechanism which improve performance. The error at the hidden nodes can still be estimated by the same method employed by backprop, but the algorithm which employs this error to govern the weight change need not be the LMS algorithm. It can be any one of a number of more sophisticated descent methods [18].

Replacing the LMS algorithm in this manner can eliminate several shortcomings associated with the standard backpropagation algorithm, shortcomings such as inconsistent training, or failing to converge in a timely manner.
A training algorithm, already formulated, which fits this class of fast algorithms is one which uses a recursive least squares estimator as the weight update mechanism [19]. The success enjoyed by this algorithm compared to backprop in terms of speed and consistency of training demonstrates the soundness of this insight.

This dissertation presents another example of a training algorithm based on this insight. This training algorithm uses the method of conjugate-gradient directions [18] to solve normal equations formulated at every node in the network. Subsequent chapters will discuss this algorithm in detail and evaluate its performance compared to backprop and the method which uses the recursive least-squares estimator as the weight update mechanism.

Several algorithms to train multilayer perceptrons employing the conjugate gradient method have been devised [20-23]. Among these are several variations devised by Johansson et al. [20], and Moller [21]. A comparative discussion of these methods and others has been done by Battiti [22]. Although these algorithms also use the method of conjugate gradient directions, there is a fundamental difference between them and the one introduced in this dissertation.

The focus of these conjugate gradient algorithms is the error surface with respect to the weights. These algorithms attempt to traverse the error surface towards a global minimum by updating the weights in directions which do not undo the error minimization
which previous update directions have achieved. Such directions are Q-conjugate where Q is the Hessian, that is, the matrix containing the second derivatives of the error surface with respect to the weights in the network. Whereas, as shall be seen in Chapter 3, the algorithm discussed in this dissertation does not approach the problem of solving the multilayer perceptron training problem by considering the error surface. Its goal is to employ the backpropagated error in the formulation of a normal equation [24] at every node and to solve this normal equation at every node using the method of conjugate-gradient directions. It is beyond the scope of this work to examine the path which this algorithm traverses along the error surface. Its path would not be expected to be composed of Q-conjugate directions.

The algorithm introduced in this dissertation requires the formulation of a normal equation [24] at every node. The next chapter discusses the normal equation and illustrates how it can be formulated using the insight that backpropagating the error and updating the weights are distinct and separate. Chapter 3 will introduce and describe in detail the example of an algorithm based on this realization and the method of conjugate gradients and illustrate the differences between it and the conjugate-gradient algorithms. Chapter 4 presents results of experiments performed with this algorithm. Chapter 5 will present a rudimentary parallel processing scheme to render the new algorithm computationally efficient. Chapter 6 comprises the conclusions of this investigation.
CHAPTER 2: THE NORMAL EQUATION AND THE SUPERVISED TRAINING OF A MULTILAYER FEEDFORWARD PERCEPTRON

The update mechanisms (besides backprop) which are the subject of this dissertation require the formulation of a normal equation at every node in the network. This chapter introduces the normal equation and explains how it relates to the supervised training of the multilayer perceptron.

2.1 The Multilayer Feedforward Perceptron and the Linear Combiner:

As mentioned before, the basic components of a feedforward multilayer perceptron are the nodes as illustrated by Figure 1.2. The node consists of a weight vector \( \mathbf{w} \), a bias \( \theta \), a linear combiner and a node nonlinearity, also known as the activation function.

A widely used activation function is the sigmoid given by

\[
z = f(y) = \frac{1 - e^{-sy}}{1 + e^{-sy}}
\]

which will be used throughout this dissertation. The parameter \( s \) specifies the steepness of the curve.

The inverse of this function is given by:

\[
y = f^{-1}(z) = \frac{1}{s} \ln \left( \frac{1 + z}{1 - z} \right)
\]
The linear combiner generates the inner product of the weight vector $w$ and the applied input vector $x$ to the node and then adds the bias $\theta$ to the inner product. If the input vector $x$ is augmented by the constant 1, then the bias $\theta$ can be included in the weight vector $w$ and the linear combiner can be considered to simply generate the inner product of $w$ and $x$, i.e., $y = x^T w$. The nonlinearity acts upon this inner product to generate $z = f(y)$ which constitutes a component in the input vector to the nodes in the next layer.

The multilayer feedforward perceptron is considered trained and will perform the desired mapping when the weight vectors of each node are such that the output of the net is within a specified tolerance of the desired outputs for a test set which is drawn at random and does not usually coincide with the training set [16]. When this happens, the output of each node in the net is such that the output of the nodes in the output layer of the net is close to the desired output. Training the net, hence, implies training each node in the net for the desired response.

Assume that the desired output $d_p'$ of a node corresponding to an input pattern $x_p$, where $p=1,2,...,M$, $M$ being the number of input patterns, is known or can be estimated. Then the desired output $d_p$ of the linear combiner is simply $f^{-1}(d_p')$.

When the linear combiner makes the correct input vector to output scalar mapping, the node will yield the correct desired response. Hence it is sufficient to concentrate attention on the training of a linear combiner which is the subject of the next section.
2.2: The Normal Equation and the Linear Combiner

Consider the linear combiner shown in Figure 2.1. The output $y$ of the combiner is the weighted sum of the components of the input vector $x$, or $x^Tw$, i.e., the inner product of the input vector and the weight vector associated with the linear combiner. The linear combiner hence performs a linear mapping between an input vector set and a set of scalar outputs.

The goal is to map a set of $M$ input vectors to a set of $M$ output scalars such that the mean squared error $E$ given by:

$$E = \sum_{p=1}^{M} \frac{1}{2} (d_p - y_p)^2$$

is a minimum.

Since $y = x^Tw$, $E$ can be expressed in terms of the weight vector $w$ as:

$$E = \sum_{p=1}^{M} \frac{1}{2} (d_p - x_p^Tw)^2$$

The weight vector which minimizes this mean squared error $E$ can be obtained by setting to zero the derivative of the mean squared error with respect to the weights [24]:

$$\frac{\partial E}{\partial w} = \sum_{p=1}^{M} (d_p x_p - x_p^T w) = 0$$

(2.3)
Figure 2.1: A Linear Combiner
Defining

\[ R = \sum_{p=1}^{M} x_p x_p^T \]

and

\[ p = \sum_{p=1}^{M} d_p x_p \]

(2.3) can be expressed in matrix form as

\[ R w = p \]  \hspace{1cm} (2.4) \]

The matrix \( R \) can be interpreted as the auto-correlation matrix of the input vector set and the vector \( p \) as the cross-correlation between the training patterns and the corresponding desired responses. The above equation is referred to as the deterministic normal equation in the context of adaptive filtering. The solution to the normal equation is the weight vector which minimizes the sum of the squared errors [24].
2.3: The LMS Algorithm

The normal equation (Eq. 2.4) derived in the preceding section can be solved for $w$ either by inverting $R$ or recursively using several established methods. One such recursive method is the gradient descent method which changes the weights in the opposite direction of the gradient of the mean squared error (mse).

For a linear combiner the function describing the error surface in terms of the weights (Eq. 2.3) is a quadratic. Hence the surface is convex and moving the weights in the opposite direction of the gradient will eventually bring them to the bottom of the bowl" -- i.e., to the weight vector which minimizes the sum of the squared error [17, 24].

The gradient at any point in weight space, i.e., the derivative of the mse with respect to the weights is given by Eq. 2.3 to be $Rw - p$. The weight change given by the gradient descent method should therefore be $-\mu(Rw - p)$, where is a suitable step-size. Updating the weights iteratively according to this rule will eventually bring the weights to the solution of Eq. 2.4, provided that $\mu$ is small enough.

The LMS algorithm was devised by Widrow and Hoff [17] to solve the linear combiner problem. The LMS update is obtained from the gradient descent update simply by using the outer product $x_p x_p^T$ as an instantaneous estimate for $R$ and $d_p x_p$ as an estimate for $p$. 
The LMS algorithm hence dictates that at every iteration the weight change be
\[-\mu (x_p x_p^T w - x_p d_p) \] or \[-\mu x_p (x_p^T w - d_p).\] The inner product \(x_p^T w\) is the actual output \(y\) of the linear combiner, so the weight change can be written as
\[
\Delta w = \mu x_p (d_p - y_p) \quad (\text{Eq. 2.5})
\]

The weight change shown in Eq. 2.5 is the product of the input vector and the difference between the desired output and actual output, i.e., the error of the linear combiner.

The quantity \(x_p (d_p - y_p)\) also happens to be the derivative of the squared error with respect to the weights. This fact will be used in the next section which will illustrate the way in which standard backprop employs the LMS update.
2.4: Backprop

To describe the derivation of standard backprop, it is necessary to establish a notation to distinguish between nodes in particular layers and their respective inputs and outputs. The reader is referred to Figure 2.2.

A particular node shall be designated by subscripts to identify the layer and the position of the node within the layer. For example, the vector \( x_{p\ell} = [x_{p\ell1}, x_{p\ell2}, ..., x_{p\ell N}]^T \), where T denotes transpose, is the input vector to every node in the \( \ell \)-th layer corresponding to the \( p \)-th pattern presented to the net. \( x_{p\ell i} \) is the \( i \)-th component of the input vector to each node in the \( \ell \)-th layer, hence it is the output of the \( i \)-th node in the \((\ell-1)\)-th layer and is multiplied by \( w_{n\ell i} \), the \( i \)-th component in the weight vector associated with the \( n \)-th node in layer \( \ell \). Associated with the \( n \)-th node in the \( \ell \)-th layer is the weight vector \( w_{n\ell} = [w_{n\ell 1}, w_{n\ell 2}, ..., w_{n\ell N}]^T \) and a bias \( \theta_{n\ell} \). \( w_{n\ell i} \) connects the \( i \)-th node in the \((\ell-1)\)-th layer to the \( n \)-th node in the \( \ell \)-th layer. \( N \) is the number of weights associated with the node and the number of nodes in the previous layer. The bias \( \theta_{n\ell} \) is clamped to 1 and is associated with the \( n \)-th node in the \( \ell \)-th layer. Every node in the same layer has the same number of weights.

The linear combiner of the node forms the sum \( y_{p\ell n} = x_{p\ell}^T w_{n\ell} + \theta_{n\ell} \). Augmenting \( x_{p\ell} \) by 1 and \( w_{n\ell} \) by \( \theta_{n\ell} \) allows \( y_{p\ell n} \) to be written in a simpler form as \( y_{p\ell n} = x_{p\ell}^T w_{n\ell} \) where \( x_{p\ell} \) and \( w_{n\ell} \) are now \( N+1 \) vectors. The output of the node will be designated \( z_{p\ell n} = f(y_{p\ell n}) \).
The input vector to every node in the next layer, the \((\ell+1)\) th layer, is then

\[
x_{p,\ell+1} = [x_{p,\ell+1,1}, x_{p,\ell+1,2}, \ldots, x_{p,\ell+1,J}]^T
\]

\[
= [z_{p1}, z_{p2}, \ldots, z_{pJ}]^T
\]

where \(J\) is the number of nodes in the \(\ell\)-th layer.

The nodes in the \((\ell+1)\)-th layer generate their outputs which are the inputs to nodes in the next layer and this continues on to the nodes in the output layer \(L\) whose outputs are the outputs of the net.
Figure 2.2: Diagram of a Multilayer Feedforward Perceptron Illustrating Notation Used
Backprop seeks to change the weights iteratively in order to minimize the mean squared error at the output of the network summed over every output node.

This output error $E_p$ corresponds to a particular pattern $p$ and is the error summed over every output node as follows:

$$E_p = \sum_{n=1}^{N} \frac{1}{2} (d_{pn} - z_{pLn})^2$$

$$= \sum_{n=1}^{N} \frac{1}{2} (d_{pn} - f(y_{pLn}))^2$$

$$= \sum_{n=1}^{N} \frac{1}{2} (d_{pn} - f(x_{pl}^T w_{Ln}))^2$$

where $n$ is the summing index over every node in the output layer, $N$ is the number of nodes in the $L$th layer and $L$ is the number of layers in the net. That is, the $L$th layer is the output layer of the net.

Thus, the backprop weight change $\Delta w_{\ell ni}$ for every node in the network and at every pattern presentation is proportional to $-\frac{\partial E_p}{\partial w_{\ell ni}}$

where

$$\frac{\partial E_p}{\partial w_{\ell ni}} = \frac{\partial E_p}{\partial y_{p/n}} \cdot \frac{\partial y_{p/n}}{\partial w_{\ell ni}}$$

The second factor is given by

$$\frac{\partial y_{p/n}}{\partial w_{\ell ni}} = \frac{\partial}{\partial w_{\ell ni}} \sum_{k=1}^{N} w_{\ell ni} x_{p//k} = x_{p/\ell}$$
Defining

\[ \delta_{p/n} = -\frac{\partial E_p}{\partial y_{p/n}} \]

the weight change is then proportional to:

\[ -\frac{\partial E_p}{\partial y_{p/n}} = \delta_{p/n} x_{p/i} \]

Hence the weight change is given by:

\[ \Delta w_{ri} = \eta \cdot \delta_{p/n} x_{p/i} \]

For nodes in the output layer \( \delta_{pLn} \) is given by:

\[ \delta_{pLn} = \frac{\partial E_p}{\partial z_{pLn}} \cdot \frac{\partial z_{pLn}}{\partial y_{pLn}} \]

\[ = (d_{pn} - z_{pLn}) f'(y_{pLn}) \]

and for nodes in the hidden layers \( \delta_{pLn} \) is derived as follows:

\[ \delta_{pLn} = -\frac{\partial E_p}{\partial y_{p/n}} = -\frac{\partial E_p}{\partial z_{p/n}} \cdot \frac{\partial z_{p/n}}{\partial y_{p/n}} = -\frac{\partial E_p}{\partial z_{p/n}} \cdot f'(y_{p/n}) \]

To obtain an expression for the first factor, we differentiate \( E_p \) with respect to the output of the node:

\[ -\frac{\partial E_p}{\partial z_{p/n}} = -\sum_k \frac{\partial E_p}{\partial y_{p,\ell+1,k}} \frac{\partial y_{p,\ell+1,k}}{\partial z_{p/n}} = \sum_k \delta_{p,\ell+1,k} \frac{\partial}{\partial z_{p/n}} \sum_i w_{\ell+1,k,i} z_{p/i} \]

\[ = \sum_k \delta_{p,\ell+1,k} w_{\ell+1,k,n} \]

where \( k \) is over nodes in the \( \ell+1 \) th layer, and in keeping with previously established notation, \( w_{\ell+1,k,i} \) connects the i-th node in the \( \ell \)-th layer to the k-th node in the \( \ell+1 \)-th layer.
Hence for nodes in the hidden layers, one has:

$$\delta_{p/n} = f'(y_{p/n}) \sum_k \delta_{p/\ell+1,k} w_{\ell+1,k,n}$$

The backprop update for weights associated with nodes in the hidden layer is thus

$$\Delta w_{\ell ni} = \eta \delta_{p\ell n} x_{p\ell}$$ with \( \delta_{p\ell n} \) given by Eq 2.7, and the backprop update for weights associated with nodes in the output layer is:

$$\Delta w_{Lni} = \eta \delta_{pLn} x_{pL}$$ with \( \delta_{pLn} \) given by Eq 2.6 [2].

Since \( f(y) \), the sigmoid activation function, is monotonically increasing, the quantity

\[(d'_p - z_p)f(y_p)\] is proportional to \( f^{-1}(d'_p) - y_p \). A comparison of the update at the output nodes with the LMS update (Eq. 2.5) illustrates the fact that the backprop weight update at nodes in the outer layer is indistinguishable from the LMS update.

Furthermore, the LMS update \((d_p - z_p)\) is also the derivative of the mean squared error with respect to the output of the linear combiner. Analogously, the quantity

$$\delta_{p/n} = -\frac{\partial E_p}{\partial y_{p/n}}$$ is the derivative of the mean squared error with respect to the output of each linear combiner associated with nodes in the hidden layer. Hence, \( \delta_{p\ell n} \) indicates the direction and amount by which the output at nodes in the hidden layers would have to change so that the error would be reduced. This quantity \( \delta_{p\ell n} \) serves as a generalized error at the output of nodes in the hidden layer.

Backprop has become the mainstay of neurocomputing [16]. It does however suffer from
shortcomings. Training with backprop can be inconsistent. The mean squared error may stay at the same level through many iterations before plummeting sharply [3]. For certain applications, parameters and initial weight sets, the error may not decrease to a tolerable level at all. When this happens, the network is said to have not converged. Failure to converge has been ascribed to the network getting stuck at a local minimum. Even when the network converges, it sometimes converges very slowly, only after the input vector set has been presented repeatedly an inordinate number of times.

 Alternatives to backprop are therefore required. An insight [9] which uncovers a spectrum of alternatives is the realization that finding the weight update $\Delta w_{t_{nl}} = \delta_{p_{fn}}x_{p_{fi}}$ and finding the backpropagated error $\delta_{p_{fn}}$ at the node are two different tasks. Treating these two functions -- estimation of error and updating of weights -- as distinct and separate has an important ramification: it allows for variations on the weight update mechanism which improve performance. The error at the hidden nodes $\delta_{p_{fn}}$ can still be estimated by backpropagating the output error back through the hidden layers of the network, but the algorithm which employs this error to dictate the weight change need not be the LMS algorithm. It can be any one of a number of more sophisticated descent methods such as recursive least squares estimation, conjugate gradient directions and others [19]. The only requirement for the update method is that it be iterative and eventually bring the weight vector associated with the node to the solution of the normal equation. The next section lays out the form which any algorithm based on this insight might take and how $\delta_{p_{fn}}$ can be used to formulate a normal equation at every node.
2.5: A Generalized Approach to the Training of the Multilayer Feedforward Perceptron:

If training a feedforward multilayer perceptron is viewed as solving the normal equation at every node in the network, then any training algorithm, including standard backprop, can be stated in the following manner:

Set all the weights and biases in the network to initial random values and repeat the following steps until the mean squared error at the output of the net is below a prescribed tolerance:

1. Choose an input pattern at random from the training set and apply it to the input of the network. Propagate the pattern through the net finding the output at every node including the output of nodes in the output layer. (Since a pattern will be presented at each iteration n, the hitherto used subscript p will be dispensed with and the input to any node at iteration n will be designated \( x_n \).)

2. Backpropagate the error 6 to every node in the network using the rules laid down in equations 2.6 and 2.7.

3. Formulate the normal equation \( Rw = p \) at every node in the following manner:

Estimate the auto-correlation matrix \( R \) at every layer in the network.

Eq. 2.3 gives \( R = \sum_{p=1}^{M} x_p x_p^T \)
An estimate for \( R \) at iteration \( n \), which does not require the presentation of the entire training set is:

\[
R(n) = \sum_{k=1}^{n} b^{n-k} x_k x_k^T
\]

where \( b \) is a decay factor so that the more recently presented patterns are more prominent.

Estimate the cross-correlation vector \( p \) at nodes in the hidden layers of the network employing the backpropagated error \( \delta \) at nodes in the hidden layers. Since \( \delta \) indicates the direction and amount by which the output at nodes in the hidden layers would have to change so that the error would be reduced, an estimate for the desired response of the node is:

\[
d = y + \mu \delta
\]

Estimate \( p \) at nodes in the hidden layers in the following manner:

\[
p(n) = \sum_{k=1}^{n} b^{n-k} d_k x_k = \sum_{k=1}^{n} b^{n-k} (y_k + \mu \delta_k) x_k
\]

The desired response at nodes in the output layer is explicitly specified at the output of the nonlinearity. It is necessary to reflect the desired response to the output of the linear combiner of the node, i.e., the input of the nonlinearity. This can be done by using the inverse function given in Eq. 2.2. If the desired response at the output of the nonlinearity is \( d' \), then the desired response at the input of the nonlinearity which is the same as the output of the linear combiner is:

\[
d = f^{-1}(d') = \frac{1}{s} \ln \left( \frac{1 + d'}{1 - d'} \right)
\]
Estimate $\mathbf{p}$ at nodes in the output layers as follows:

$$
\mathbf{p}(n) = \sum_{k=1}^{n} b^{n-k} d_k \mathbf{x}_k = \sum_{k=1}^{n} b^{n-k} \mathbf{x}_k \left\{ \frac{1}{s} \ln \left( \frac{1 + d'}{1 - d'} \right) \right\}
$$

Update $\mathbf{R}$ and $\mathbf{p}$ iteratively at every iteration, or pattern presentation, as follows:

$$
\mathbf{R}(n) = b \mathbf{R}(n-1) + \mathbf{x}_n \mathbf{x}_n^T \quad \text{(Eq. 2.8)}
$$

$$
\mathbf{p}(n) = b \mathbf{p}(n-1) + d_n \mathbf{x}_n \quad \text{(Eq. 2.9)}
$$

In the case of the algorithm based on the method of recursive least squares, the autocorrelation matrix $\mathbf{R}$ is not estimated. Instead a quantity which conveys the same information, i.e., the inverse of the autocorrelation matrix, $\mathbf{R}^{-1}$, is computed at every layer. The matrix inversion lemma [18] can be used to show that $\mathbf{R}^{-1}$ can be iteratively estimated by first calculating a vector known as the Kalman gain vector $\mathbf{k}$ (also at every layer) as follows:

$$
\mathbf{k}(n) = \frac{\mathbf{R}^{-1}(n-1) \mathbf{x}_n}{b + \mathbf{x}_n^T \mathbf{R}^{-1}(n-1) \mathbf{x}_n}
$$

The Kalman vector $\mathbf{k}$ very cleverly and succinctly summarizes the history of all the patterns presented to the layer. The inverse of the autocorrelation matrix $\mathbf{R}^{-1}$ can then be expressed in terms of the Kalman gain vector $\mathbf{k}_n$, the pattern presented to the layer $\mathbf{x}_n$ and the forgetting factor $b$ as follows:

$$
\mathbf{R}^{-1}(n) = \frac{\mathbf{R}^{-1}(n-1) - \mathbf{k}_n \mathbf{x}_n^T \mathbf{R}^{-1}(n-1)}{b}
$$
In the case of backprop the estimates used for $R$ and $p$ are:

$$R(n) = x_n x_n^T \quad \text{and} \quad p(n) = (y_n + \mu \delta_n)x_n,$$

where $\mu$ is the learning rate.

4. Update the weights towards values which will satisfy the normal equation. The choice of update mechanism at this step makes for a particular training algorithm. Standard backprop employs the LMS update here. The weight vector $w^*$ which is the solution to the normal equation is $w^* = R^{-1} p$. The recursive least squares method has been estimating the inverse of the autocorrelation matrix $R^{-1}$ all along. Hence it sets the weights equal to $R^{-1} p$. But Scalero [19] showed that setting the weights equal to $R^{-1} p$ is equivalent to updating the weights for nodes in the output layer by $\Delta w_n = k(n)[d_n - y_n]$, and updating the weights for nodes in the hidden layer by the quantity $\Delta w_n = \mu k(n) \delta_n$. The crosscorrelation vector $p$ is implicitly included in this update and need not be explicitly computed when training with the recursive least squares based algorithm.

The details of the conjugate-gradient based algorithm are deferred to Chapter 3. Briefly this update is made in a direction $d_n$ which depends on the quantity $R w - p$ and the previous weight update. The step-size $\alpha$, i.e., how far the weight change is made in the given direction depends on the set $d_i$ of directions along which the update is made.

Experimentation shows a qualitative difference in performance between the method of conjugate gradients and recursive least squares estimation in various pattern recognition problems (see Chapter 4).
The possibilities for update mechanisms have not been exhausted and different update rules might better suit different applications.

In the next chapter, an algorithm which uses the method of conjugate gradient directions to solve the normal equation at each node will be introduced and described. A characteristic of the method of conjugate gradients which would lead one to expect faster convergence than backprop is that if $R$ and $p$ are known exactly and not estimated, the method of conjugate gradients would find the solution to the normal equation in exactly the same number of steps as there are weights in the node [18]. The training of a multilayer feedforward perceptron is, however, a more complicated situation because neither $R$ nor $p$ are known but have to be estimated as training proceeds. During the initial phases of training the estimates of $R$ and $p$ are not very good, but as training continues, these estimates improve and training accelerates.
CHAPTER 3: TRAINING ALGORITHM WITH NORMAL EQUATION
SOLVED BY THE METHOD OF CONJUGATE GRADIENTS

This chapter details a new training algorithm which employs the method of conjugate directions to solve the normal equation at every node. The method of conjugate directions can solve the normal equation in n steps where n is the number of weights in the node. A description of the method is first given.

3.1: The Method of Conjugate-Gradients to Solve the Normal Equation

The normal equation to be solved is:

\[ \mathbf{Rw} = \mathbf{p} \]  \hspace{1cm} (3-1)

A set of n vectors \( \mathbf{d}_i \) are said to be \( \mathbf{R} \)-conjugate if \( \mathbf{d}_i^T \mathbf{R} \mathbf{d}_j = 0 \), for \( i \neq j \)

Such a set of vectors is linearly independent and forms a basis for the n-space.

Hence the move from an initial weight vector \( \mathbf{w}_o \) to the solution \( \mathbf{w}^* \) of the above equation can be expressed in terms of this basis as:

\[ \mathbf{w}^* - \mathbf{w}_o = \sum_{i=1}^{n} \alpha_i \mathbf{d}_i \]  \hspace{1cm} (3-2)

The \( \mathbf{R} \) conjugacy of the directions \( \mathbf{d}_i \) allow the coefficients \( \alpha_k \), to be found in the following manner:

\[ \mathbf{d}_k^T (\mathbf{Rw}^* - \mathbf{Rw}_o) = \alpha_k \mathbf{d}_k^T \mathbf{R} \mathbf{d}_k \]  \hspace{1cm} (3-3)

Since \( \mathbf{Rw}^* = \mathbf{p} \), Equation 3-3 becomes

\[ \mathbf{d}_k^T (\mathbf{p} - \mathbf{Rw}_o) = \alpha_k \mathbf{d}_k^T \mathbf{R} \mathbf{d}_k \]  \hspace{1cm} (3-4)
Also since

\[ w_k = w_o + \sum_{i=1}^{k} \alpha_i d_i \]  \quad (3-5)

we have \( d_k^T R w_k = d_k^T R w_o \) because the directions are \( R \) conjugate.

Therefore Equation 3-4 can be written as

\[ d_k^T (p - Rw_k) = \alpha_k d_k^T R d_k \]  \quad (3-6)

Since \( Rw - p \) is the gradient to the quadratic error surface (3-6) we have

\[-g_k^T d_k = \alpha_k d_k^T R d_k \]  \quad (3-7)

which yields

\[ \alpha_k = -\frac{g_k^T d_k}{d_k^T R d_k} \]  \quad (3-8)

It can be shown [19] that choosing the directions \( d_i \) in the following way ensures that they will be \( R \) conjugate:

\[ d_o = -g_o \]

for \( k=0 \) to \( n-2 \)

\[ d_{k+1} = -g_{k+1} + \beta_k d_k \]

where

\[ \beta_k = \frac{g_{k+1}^T R d_k}{d_k^T R d_k} \]
The entire method can be summarized as follows:

a. Set \( k = 0 \), select an initial random weight vector \( \mathbf{w}_0 \).

b. Find \( \mathbf{g}_o = \mathbf{Rw}_o - \mathbf{p} \). If \( \mathbf{g}_o = 0 \), stop, else set \( \mathbf{d}_o = - \mathbf{g}_o \).

c. \[ \alpha_k = -\frac{\mathbf{g}_k^T \mathbf{d}_k}{\mathbf{d}_k^T \mathbf{Rd}_k} \]

d. \( \mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{d}_k \)

e. If \( k = n - 1 \), stop, else continue

f. \( \mathbf{g}_{k+1} = \mathbf{Rw}_{k+1} - \mathbf{p} \). If \( \mathbf{g}_{k+1} = 0 \), stop.

g. \[ \beta_k = -\frac{\mathbf{g}_{k+1}^T \mathbf{Rd}_k}{\mathbf{d}_k^T \mathbf{Rd}_k} \]

h. \( \mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{d}_k \)

i. Increment \( k \) and go to c.

The weight vector which results upon the completion of the above \( n \) steps is \( \mathbf{w}^* \) the solution to the normal equation [18], [20].
3.2: The Training Algorithm.

The algorithm outlined in this section uses the method of conjugate gradients to solve the normal equation formulated at every node in the feedforward multilayer perceptron. This algorithm falls into the category of the general training algorithm presented in Chapter 2. Training starts with every weight and bias in the network initialized to random values. Training patterns are presented to the net and the corresponding desired responses are used to estimate the errors which are backpropagated to the nodes in the hidden layers.

The autocorrelation matrix $\mathbf{R}$ is updated at every layer using the rule $\mathbf{R}_k = \mathbf{R}_{k-1} + b\mathbf{x}_k\mathbf{x}_k^T$ and the crosscorrelation vector $\mathbf{p}$ is updated at each node using the rule $\mathbf{p}_k = \mathbf{p}_{k-1} + b\mathbf{x}_kd'_k$. The quantity $d'_k$ is an estimate of the desired output, if the node is in the hidden layer, and is the actual desired output of the linear combiner if the node is in the output layer.

Step 3 of the generalized training algorithm takes the form of a conjugate gradient direction update in the case of this algorithm. At first glance it would appear that the n steps would have to be taken every time the autocorrelation matrix $\mathbf{R}$ and the crosscorrelation vector $\mathbf{p}$ were updated. This would not be conducive to an iterative method of training. Experimentation bears out that each one of the steps can be taken with each presentation of an input vector since $\mathbf{R}$ does not change greatly from one pattern presentation to another. Of course the $\mathbf{R}$ and $\mathbf{p}$ are, in the case of a multilayer perceptron, only estimates and therefore the training of the multilayer perceptron will not
be completed in n steps.

In order to make the update the direction $d_n$ for the node is calculated as stated in section 3.1, using the current estimates of $R$ and $p$, the current weight vector of the node $w$ and the previous direction. The step-size $a$ is then calculated as stated in section 3.1 and the weight update $\Delta w_p = ad_n$ is made.

The entire training algorithm can be summarized as follows:

1. Initialize all the weights in the network to random values.

2. Propagate a training pattern through the net, finding the output at every node given by $f(y) = \frac{1 - e^{-s y}}{1 + e^{-s y}}$, where $s$ is the slope of the nonlinearity, and $y$ is the output of the linear combiner of every node $y = \sum_{i=1}^{n} w_i x_i$ with $n$ being the number of weights in the node.

3. Backpropagate error signals:

   Compute the derivative of $f(y)$ using
   $$f'(y) = \frac{s}{2} \left( 1 - f^2(y) \right)$$

   Calculate error signals in the output layer, where the layer index is $\ell = L$, by evaluating
   $$\delta_{pL} = (d'_{pn} - z_{pL}) f''(z_{pL})$$

   for every node $n$ in the output layer, where $d'_{pn}$ is the desired response at the output of the nonlinearity.
For the hidden layers, starting at layer $\ell = L-1$ and decrementing through $\ell = 1$, find error signals by solving

$$\delta_{pfn} = f'(y_{pfn}) \sum_k \delta_{p,f+1,k} w_{f+1,k,n}$$

(2.7)

for every node in the 8th layer, where $k$ runs over nodes in the $(\ell+1)$-th layer.

4. Find the desired summation output $d_n$ for the linear combiner in each node in the layer using $d_n = f^{-1}(d'_n) = \frac{1}{\ln s} \ln \left( \frac{1 + d'_n}{1 - d'_n} \right)$ for the nodes in the output layer, and using $d_n = y_n + \mu \delta_{ln}$ for the nodes in the hidden layer.

5. Update the autocorrelation matrix for each layer

$$\mathbf{R}(n) = b \mathbf{R}(n-1) + \mathbf{x}_n\mathbf{x}_n^T$$

Update the cross-correlation vector for each node in the following manner:

$$p(n) = b p(n-1) + d_n \mathbf{x}_n$$

where $n$ is the pattern presentation index.
6. Update the weight vector for every node in the network in the following manner:

At every node calculate

i) \( g_n = R_n w_n - p_n \)

If \( g_n = 0 \), do not update the weight vector for the node, else perform the following steps:

ii) Find the direction \( d_n \). If the iteration number is an integer multiple of the number of weights in the node then \( d_n = -g_n \), else \( d_n = -g_n + \beta_n d_{n-1} \) where

\[
\beta_n = -g_n^T \frac{R_{n-1} d_{n-1}}{d_{n-1}^T R_{n-1} d_{n-1}}
\]

iii) Find the step-size \( \alpha_n = -\frac{g_n^T d_n}{d_n^T R_n d_n} \)

iv) Modify the weight vector:

\[
w_{n+1} = w_n + \alpha_n d_n
\]

7. Test for completion. If the network has not yet converged, go back to step 2.
3.3: Other Conjugate-Gradient Training Algorithms:

Other conjugate gradient algorithms to train multilayer perceptrons have been devised [20-23]. The underlying philosophy of these algorithms differs greatly from that of the algorithm just presented in 3.2. This algorithm trains the network by using the backpropagated error to formulate a normal equation at every node and by solving these equations with the method of conjugate-gradient directions.

The conjugate-gradient algorithms discussed in the literature operate under the assumption that near the solution, the error as a function of all the weights in the network, given by \( f(w) = \frac{1}{2P} \sum_{p=1}^{P} E_p \), approximates a quadratic function. This can been seen from its second-order Taylor approximation: \( f(w) = \frac{1}{2} w^T Q w - b^T w \), where

\[
E_p = \sum_{n=1}^{N} \frac{1}{2} (d_{pn} - z_{pln})^2 , \quad P \text{ is the number of patterns in the training set}, \quad N \text{ is the number of output nodes}, \quad d_{pn} \text{ are the desired outputs and } z_{pln} \text{ are the actual outputs}, \quad w \text{ represents all the weights in the network}. \quad Q \text{ is the matrix of second partial derivatives: the Hessian.}
\]

It is a square matrix of the same dimension as the number of weights in the network. These algorithms seek to find a system of Q-conjugate directions along which to traverse the error surface. Since Q, that is, the Hessian is so large and cannot be practically computed, the main problem faced by such algorithms is to find the Q-conjugate directions and the appropriate step-sizes without explicitly using the Hessian.
Johansson, Dowla and Goodman [20] eliminate the necessity of computing the Hessian
by using a line search to determine the step-size $\alpha$, ($\alpha$ minimizes $f(w + \alpha d)$) and various
formulae which do not explicitly use $Q$ for the directions. The weights are updated along
the conjugate gradient direction $d$, the entire training set is passed through the network,
$f(w + \alpha d)$ is evaluated, $\alpha$ is changed according to the line-search procedure until it
satisfies the minimization criteria. This line-search involves passing the training set
several times through the network to evaluate $f(w+\alpha d)$, before deciding on the weight
change[33].

Moller [21] dispenses with the line-search procedure by approximating second-order
information using the first-order errors backpropagated through the network. An update
still requires the presentation of the entire training set through the network.

These methods cannot be employed in time-critical applications that require on-line
training since an update cannot be made with the presentation of each pattern.

Johansson et al. report better performance than backprop on parity problems. The metric
they used was the number of function evaluations. The version of backprop used in their
experiments, however, was what they termed "offline backprop" where an update was
made only after the presentation of the entire training set. Backprop is known to perform
better when an update is made with each pattern presentation [9] and indeed this version
of backprop performed better than the conjugate gradient algorithm devised by Johansson
et al. at experiments conducted by this author with pattern recognition (the weakly and
strongly correlated pattern classifications discussed in chapter 4). In these problems, backprop achieved the same mean squared error in less than 80% of number of function evaluations as Johansson's algorithm.

The algorithm introduced in this dissertation makes no assumption about the error surface. Its goal is to make better use than backprop of the first-order information present in the backpropagated errors. Henceforth, it shall be referred to as the conjugate-gradient based algorithm as opposed to conjugate-gradient algorithms which expressly attempt to traverse the error surface in Q-conjugate directions. It is beyond the scope of this work to examine the path which the conjugate-gradient based algorithm traverses along the error surface. Its path would not be expected to be composed of Q-conjugate directions.
CHAPTER 4: EXPERIMENTAL RESULTS

As mentioned in the introduction, the multilayer feedforward perceptron fits the classification paradigm. Hence, the experiments chosen to test the performance of the conjugate-gradient based method are pattern classification problems.

By way of comparison, this chapter evaluates the performance for three classification problems by the three training algorithms discussed in this dissertation: standard backpropagation, the method which uses the recursive least squares estimator [18] (discussed in section 2.5), and the newly devised algorithm which uses the method of conjugate gradient directions to solve the normal equation at every node.

The form of the backpropagation algorithm used is that which is most commonly used in industry and which enjoys the fastest convergence: backpropagation with momentum. The weight update consists of the one given in Eq. 2.8 and Eq. 2.9, and a small fraction of the previous update: $\eta \Delta w_{i-1}$, where $\eta$ is known as the momentum factor.

Before comparisons about performance could be made, the following had to be determined: the architecture adequate for the solution of the particular problem, the parameters governing the training, that is, the decay factor $b$ and step-size $\mu$ in the case of the recursive least squares algorithm and the new algorithm, and learning rate $\mu$ and momentum $\eta$ in the case of backprop.
To find the architecture which was sufficient for the solution of each problem, the number of nodes in the hidden layer was first set at four. The network was initialized to a fixed weight set and then trained using the recursive least squares algorithm which has been shown to display fast convergence [19]. The parameters used were the same as those used by Scaler, step-size $\mu = 40$, decay factor $b = 0.99$. Initial experimentation indicated that 0.2 as a choice for the slope of the nonlinearity limited the number of times training was unsuccessful. This was the slope used in the case of all these experiments. The error was examined at every 50th iteration. The maximum number of iterations was 500. The number of nodes in the hidden layer was progressively increased from 4 until error decreased as training went on. Once the minimum number of nodes in the hidden layer was determined, the network was trained by each of the algorithms with the results being averaged over a hundred initial weight sets. The parameters were varied through the ranges shown in Table 1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Initial value</th>
<th>Final value</th>
<th>Increments</th>
</tr>
</thead>
<tbody>
<tr>
<td>step-size</td>
<td>20</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>learning rate</td>
<td>0.1</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td>momentum</td>
<td>0.1</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td>decay factor  $b$</td>
<td>0.90</td>
<td>0.99</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 1: Range of Parameters Used in Experiments
At first the momentum factor and the decay factor were kept fixed to 0.1 and 0.9 respectively, while the mse vs. iteration curves were generated for changing step-size and learning rate. Once the best value for step-size and learning rate were determined, mse vs. iteration curves were generated for different values of b and η.

The three experiments are described and the results are laid out in the following sections. The curves shown are for that combination of parameters for which the particular algorithm yielded the best convergence.

4.1: Circle in the Square.

The following classification example illustrates the difference in training times of the standard backpropagation algorithm and the algorithms based on the recursive least squares estimator and the method of conjugate gradients. It was found that a minimum of nine nodes in the hidden layer are needed to solve this problem. The network is taught to identify co-ordinates within a circle of radius 0.3 centered within a unit square.

The input to the network consists of a two-dimensional vector representing x and y co-ordinates between -0.5 and +0.5 respectively. The output of the network consists of one node whose value should be -0.5 if the distance from the origin (the center of the square, represented by x=0.0 and y=0.0) of the input point presented to the network is greater than 0.3. That is the output node should be ‘off’ if the point lies outside the circle of radius 0.3. The output should attain a value of +0.5 if the distance from the origin of the input
point presented to the network is less than 0.3.

i.e., the output node should be "on" if the input vector lies within the circle of radius 0.3.

All three algorithms were started off from the same random initial weight set and patterns were presented in the same order. The initial weight values were uniformly distributed between -5 and +5.

For this problem the recursive least squares algorithm and the conjugate-gradient based algorithm displayed their best performance for a choice of decay factor $b=0.99$ and step-size $\mu = 40$. Backprop formed the circle fastest for a choice of $\eta = 0.9$ and a learning rate of $\mu = 0.8$.

Figures 4.1 shows the output of the network when trained with each algorithm after 50, 100, 300, 500 and 1000 iterations. It can be seen that a discernible circle starts to form after as few as 50 pattern presentations when the network is trained using the new algorithm which uses the method of conjugate gradients as the update mechanism. Backprop requires at least 1000 pattern presentations to train the network until a discernible circle is formed. The algorithm based on the method of recursive least squares yielded a circle in a 100 pattern presentations.
Figure 4.1: Performance of the three Algorithms for the Circle in the Square Problem.
4.2: Weakly Correlated Patterns Classification:

Figure 4.2 shows sixteen 7x7 weakly correlated patterns which the neural network was trained to classify. Each pattern is represented by a 49 binary component vector. A value of 0.5 represents an "on", of -0.5 represents an "off".

The network, upon being trained, is expected to classify each pattern into one of 16 categories. The output of the network consists of 4 nodes. Each pattern, during the training phase, will be mapped to a 4-bit binary vector, from 0000 to 1111. The minimum number of nodes in the hidden layer is 16. The network is considered converged when it makes the correct input/output mapping. For this particular problem the correct input output mapping corresponds to a mean squared error of a value at or below 0.25.

The network was trained using the three training algorithms: backprop, the algorithm based on the recursive least squares estimator and the algorithm based on the method of conjugate-gradients. For this problem, the recursive least squares algorithm and the conjugate-gradient based algorithm displayed their best performance for a choice of decay factor b=0.99 and step-size $\mu = 400$. Backprop performed best for a choice of $\eta = 0.9$ and a learning rate of 0.8.

The mean squared error averaged over the 100 initial weight sets at every 10 iterations is shown in Figure 4.3.
Figure 4.2 Weakly Correlated Patterns
Figure 4.3 Mean-squared Error vs. Iteration for Weakly Correlated Patterns
It can be seen from Figure 4.3 that the algorithm which displayed the best performance on this particular classification problem is the one based on the method of conjugate gradients. The network performed the correct classification of the 16 weakly correlated patterns within 120 iterations when trained with this algorithm. When trained by the algorithm based on the recursive least squares, the network trained within 300 iterations. Standard backprop required 350 pattern presentations before the correct input/output mapping was made by the network.

This problem is a rather easy one. The next experiment, where the network is expected to classify patterns which are closely related to one another, is more difficult. It was conducted to provide a challenge to the algorithms and for the sake of completeness.
4.3: Strongly Correlated Patterns Classification:

Figure 4.4 shows sixteen 7x7 strongly correlated patterns which the neural network was trained to classify. Each pattern is represented by a 49 binary component vector. A value of 0.5 represents an "on", of -0.5 represents an "off". All the patterns resemble each other to a great extent, that is, they are strongly correlated. The network, upon being trained, is expected to classify each pattern into one of 16 categories. The output of the network consists of 4 nodes. Each pattern, during the training phase, will be mapped to a 4-bit binary vector, from 0000 to 1111. The architecture which is sufficient to solve this problem consists of one hidden layer with 16 nodes. The network is considered converged when the mean squared error falls to a value at or below 0.25. It is at this mse that the correct input/output mapping is made.

The network was trained using the three training algorithms: backprop, the algorithm based on the recursive least squares estimator and the algorithm based on the method of conjugate-gradients. The training runs were ensembled over a hundred initial weights sets each uniformly distributed between -0.5 and +0.5. The algorithms displayed best performance for the choice of the following parameters: a forgetting factor $b=0.99$ and $\mu=400$ for the recursive least squares method and the conjugate-gradient based method, and learning rate $\mu=0.2$ and momentum factor $\eta=0.9$ for backprop. The mean squared error averaged over the 100 ensembles at every 10 iterations is shown in Figure 4.5.
Figure 4.4 Strongly Correlated Patterns
Figure 4.5 Mean-squared Error vs. Iteration for Strongly Correlated Patterns
It can be seen from Figure 4.5 that the algorithm which displayed the best performance at classifying strongly correlated patterns was the one based on the method of recursive least squares. The network performed the correct classification of the 16 strongly correlated patterns within 300 iterations when trained with the recursive least squares algorithm. When trained by the algorithm based on the method of conjugate gradients, the network trained within 450 iterations. Standard backprop required 1000 pattern presentations before the correct input/output mapping was made by the network. Moreover the mse displayed oscillatory behavior when the network was trained using backprop. This behavior is very apparent despite the fact that the mse graphed is averaged over a 100 initial weight sets.
4.4: Computational Considerations

The new algorithm converges in less pattern presentations than backprop. However it also incurs a higher computational cost per update.

For a three-layer network with $n_i$ input nodes, $n_h$ hidden nodes and $n_o$ output nodes the number of multiplications and additions incurred by each algorithm at every iteration is shown in Table 2.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of multiplications/iteration</th>
<th>Number of additions/ iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backprop</td>
<td>$n_h (3n_i + 4n_o + 5) + 5n_o$</td>
<td>$n_h (3n_i + 4n_o + 1) + 2n_o$</td>
</tr>
<tr>
<td>New</td>
<td>$n_h(n_i^2 + 9n_i + n_o + 10) + n_h(2n_h^2 + 7n_h + 10)$</td>
<td>$n_h(2n_i^2 + 9n_i + n_o + 2) + n_o(2n_h^2 + 8n_h + 3)$</td>
</tr>
</tbody>
</table>

Table 2. Number of operations per iteration for a 3-layer network.
Table 3 shows the number of operations performed by the new algorithm and backprop respectively to reach an mse of 0.1 for each of the three test problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Backprop</th>
<th>New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of itrs required to reach an mse of 0.1</td>
<td>total # of mults. required to reach an mse of 0.1</td>
</tr>
<tr>
<td>Circle in the square</td>
<td>1000</td>
<td>140,000</td>
</tr>
<tr>
<td>Weakly correlated patterns</td>
<td>350</td>
<td>947,800</td>
</tr>
<tr>
<td>Strongly correlated patterns</td>
<td>1100</td>
<td>2,978,800</td>
</tr>
</tbody>
</table>

Table 3. Operation count for backprop and the new algorithm

In the case of the circle in the square problem, the new algorithm is competitive with backprop even on a sequential computer. With the other two problems, however, more operations have to be performed, even though training occurs in less than half the pattern presentations.

In many time-critical applications, such as adaptive equalization, controllers, and robotics, where a continuous adaptation to a stream of input signals is desired [22], online training is necessary and there is an obvious advantage to using a method which will learn in fewer iterations. The added computational complexity can be handled by parallel processors. A parallel processing scheme to render the conjugate-gradient based algorithm computationally efficient and competitive with a parallelized form of backprop is presented in the next chapter.
CHAPTER 5: A PARALLEL PROCESSING SCHEME TO IMPLEMENT THE
CONJUGATE-GRADIENT BASED ALGORITHM

This chapter presents a parallel processing scheme to render the conjugate-gradient based algorithm computationally efficient. Math co-processors and vector processors that can generate vector products as quickly as an ordinary processor generates scalar products are becoming increasingly cheaper and more available. The scheme proposed in this section will not be unduly difficult to realize, since the instructions to be carried out are few and predictable and there is no ambiguity about the timing of the available data.

5.1: Parallelizing the Conjugate-Gradient Update:

At first glance the steps shown in Chapter 3, step 6 (i to iv), appear strictly sequential. However, they are presented in this fashion simply to aid human understanding. Algebraic simplification (shown in Appendix A) yields an update which incorporates terms dependent on:

1) quantities calculated at the previous iteration,
2) quantities depending on the pattern presented to the node,

and

3) quantities depending on the backpropagated error $\delta$ to the node.
The weight change at every node and at iteration $k$ can be written in the following form:

$$
\Delta w_k = \alpha_k d_k = \frac{n_3 \delta_k^3 + n_2 \delta_k^2 + n_1 \delta_k + n_0}{\eta_2 \delta_k^2 + \eta_1 \delta_k + \eta_0}
$$

(5.1)

where

$$n_3 = \mu^3 (x_k^T c_{k-1} \cdot x_k^T d_{k-1} - x_k^T x_k) [x_k - (x_k^T c_{k-1}) d_{k-1}]$$

$$n_2 = \mu^2 b [x_k^T [2h_k - h_k^T e_{k-1} d_{k-1}] [x_k - x_k^T e_{k-1} d_{k-1}] + (x_k^T e_{k-1} x_k^T d_{k-1} - x_k^T x_k) [h_k^T e_{k-1} d_{k-1} - h_{k-1}]]$$

$$n_1 = \mu b^2 [h_k e_{k-1} [x_k^T e_{k-1} d_{k-1} - x_k^T] + x_k^T [2h_k - h_k^T e_{k-1} d_{k-1}] [h_k^T e_{k-1} d_{k-1} - h_{k-1}]]$$

$$n_0 = b [h_k^T e_{k-1} [h_k e_{k-1} - h_k^T e_{k-1} d_{k-1}]] R_k [x_k^T - x_k^T e_{k-1} d_{k-1}]$$

$$\eta_2 = \mu^2 [x_k - x_k^T e_{k-1} d_{k-1}]^T R_k [x_k - x_k^T e_{k-1} d_{k-1}]$$

$$\eta_1 = b [h_k e_{k-1} - h_k^T e_{k-1} d_{k-1}]^T R_k [h_k - h_k^T e_{k-1} d_{k-1}] + 2b [x_k - x_k^T e_{k-1} d_{k-1}]^T R_k [h_k - h_k^T e_{k-1} d_{k-1}]$$

$$\eta_0 = b [h_k^T e_{k-1} - h_k^T e_{k-1} d_{k-1}] R_k [h_k^T e_{k-1} - h_k^T e_{k-1} d_{k-1}] + \mu [x_k - x_k^T e_{k-1} d_{k-1}]$$

$$g_k = R_k w_k - p_k$$

$$c_{k-1} = \frac{R_{k-1} d_{k-1}}{d_{k-1}^T R_{k-1} d_{k-1}}$$

$$h_{k-1} = g_{k-1} + R_{k-1} \Delta w_{k-1}$$
5.2: The Parallel Processing Scheme

The weight update in Eq. 5.1 is seen to depend on quantities calculated at the previous iteration, quantities depending on the pattern presented to the node and quantities depending on the backpropagated error $\delta$ to the node.

The quantities which depend on the pattern can be computed by an additional pattern processor at the same time as the pattern is being propagated forward through the net by a network processor. Quantities which require information available at the previous iteration can be computed before and while the current pattern is being presented and made available by a previous information processor. The previous information processor uses current information in computing the quantities it shall make available at the next iteration.

The network processor also backpropagates the error $\delta$ to the node. By the time this backpropagated error is made available to the node, the other quantities shown in Eq. 5.1 will have been computed. The weight change can then be made in three computation cycles as illustrated in Figure 5.1.

The scheme calls for four processors (either per node or per layer or per network):

1. a processor to generate quantities needed by the update on the next iteration designated the Previous Iteration Processor.
2. a processor to generate quantities depending on the pattern presented to the node designated the Pattern Processor.

3. a processor to generate the quantities which shall multiply the various powers of the backpropagated error \( \delta \) designated the Update Processor.

4. a processor to propagate the patterns forward through the net and backpropagate the error \( \delta \) backward through the net designated the Network Processor.

The quantities calculated at the previous iteration and those depending on the pattern presented to the node can be computed by their respective processors and kept ready by the time the error \( \delta \) is backpropagated to the node.

Figure 5.2 shows the data flow while all the necessary quantities are being calculated. A similar parallel processing scheme for standard backprop involves two processors: the update processor and the network processor. Figure 5.3 illustrates the computational cycles necessary to effect the backprop update. It can be seen that the backprop update is ready in two computation cycles.
Figure 5.1: Computation of the Conjugate-Gradient Weight Update
Figure 5.2: Data Flow among the Processors during Computation of the Conjugate Gradient Update.
Figure 5.3: Computation of the Backprop Weight Update
Under this scheme, the algorithm based on the method of conjugate gradient directions takes 1.5 times longer than standard backprop for each update to be completed. In the experiment involving classification of the strongly correlated patterns, the algorithm based on the method of conjugate gradients converged 2.2 times faster than backprop. There would still be a time saving of a factor of 1.5. Moreover backprop did not approach the low mean squared error in a stable or smooth fashion. In matters of consistency of training, the method of conjugate gradient directions still displays superiority over standard backpropagation. In the other two experiments circle in the square and classification of weakly correlated patterns, the time-saved factors are 13 and 1.7 respectively.

Custom-made logic designed to generate the sum, product and quotient of the quantities in the conjugate weight update (Eq.5.1) simultaneously can allow the algorithm based on the method of conjugate gradient directions to make one update in as much time as backprop performs one update. With the increasing use of the multilayer feedforward perceptron in industry, it might be deemed feasible to invest in such design.
CHAPTER 6: CONCLUSIONS

This work is based on the useful concept that distinguishing between the function of estimating an error at the nodes in the hidden layers of a feedforward multilayer perceptron and the function of updating the weights makes possible a class of fast algorithms to train feedforward multilayer perceptrons.

Preliminary evidence supporting this idea was offered by the success of the algorithm devised by Robert Scalero [18] which implicitly separates the two functions of backpropagating the error and updating the weights and employs a recursive least squares estimator as the weight update mechanism.

Another algorithm within this class and developed in this dissertation was the algorithm which uses the method of conjugate-gradient directions as the update mechanism. This algorithm is outlined in Chapter 3 and the experiments performed on it, with the results listed in Chapter 4, show that it enjoys a training speed which is comparable to that based on the method of recursive least squares.

Chapter 2 elaborates on this concept and explains how the backpropagated error can be used to formulate a normal equation at every node in the multilayer feedforward perceptron. The choice of method to solve this normal equation makes for a particular training algorithm. From this point of view, standard backpropagation also fits in this
category. It uses the method of least mean squares as the weight update mechanism. This view thus serves the purpose of generalising the training problem. There are qualitative differences between each algorithm. Different choices of updating mechanisms might better suit different applications. These choices are available to the user of the neural network for a particular application.

Chapter 5 lays out a parallel processing scheme which exploits the inherent parallelism and fast convergence properties of the conjugate-gradient based algorithm. A considerable time-saving can be made even if ordinary processors instead of vector-processors are used in this scheme.

Some interesting points become apparent upon examination of the weight updates of the three algorithms. Standard backprop updates every node in the same layer along the direction of the pattern presented to the node. The backpropagated error to each node dictates how far the update will be in this direction and whether it will be followed in a positive or a negative sense, but basically every node in the same layer moves according to the same vector.

The algorithm based on the method of recursive least squares also updates nodes in the same layer along the same vector with the backpropagated error serving the same purpose as in the case of backprop. However, the recursive least squares update summarizes the entire history of the patterns presented to the layer. This explains, in a qualitative manner, why moving along the direction dictated by this update rule brings about faster
convergence than moving along the direction of the instantaneously presented pattern
dictated by the least mean squares update.

The method of conjugate directions dictates a different direction for every node in the
network, irrespective of which nodes share a common layer. This direction is determined
not only by the history of patterns presented to the node but also on the history of errors
backpropagated to the node. In this sense, the conjugate gradient algorithm utilizes more
information and displays a greater degree of freedom than the other two update rules.
This might explain why it shows superior performance for certain problems.

A consequence of estimating and utilizing the autocorrelation matrix at every layer and
cross-correlation vector at every node is that training with the conjugate-gradient based
algorithm and the recursive least squares based algorithm makes the most efficient use of
information gleaned from the knowledge of the desired response. This property of such
algorithms might make them most useful in the area of training recurrent networks [25].
Recurrent networks are trained to perform spatio-temporal mappings, i.e., mapping a set
of input and output vectors occurring in time in a particular sequence to a similar set.
They are used in control applications. These networks are usually trained with a variant
of the backpropagation algorithm, with the error being defined in a similar way. A factor
which slows such training down is that the desired response is not available on every
training run. To account for this, an update is not made at every pattern presentation, but
the backpropagated error is accumulated before the weights are updated. This is termed
batch-processing. Batch-processing is also used in some cases while training a
feedforward network but it is generally not as fast as updating at every iteration if the network being trained is a feedforward network [9]. Since the desired response while training a recurrent network is so sparsely available, any algorithm which makes the most efficient use of the available information should display better results than an algorithm based on standard backprop. The conjugate-gradient based algorithm and the recursive least squares based algorithm indeed make efficient use of information gleaned from the knowledge of the desired response and hence an adaptation of these methods to the problem of training a recurrent network might prove to be a fruitful venture.

As mentioned in the introduction, all possibilities for the update mechanism have not been exhausted and the door is open for the formulation of several such algorithms based on the realization that the estimation of the error and the update of the weights are distinct and separate.
A vector $\mathbf{g}_k$ at iteration $k$ is defined as follows:

$$\mathbf{g}_k = \mathbf{R}_k \mathbf{w}_k - \mathbf{p}_k$$  \quad (A.1)

But the autocorrelation matrix $\mathbf{R}_k$ and the crosscorrelation vector $\mathbf{p}_k$ are estimated iteratively as:

$$\mathbf{R}_k = b \mathbf{R}_{k-1} + x_k x_k^T$$

$$\mathbf{p}_k = b \mathbf{p}_{k-1} + x_k (y_k + \mu \delta_k)$$

Hence Eq (A-1) becomes

$$\mathbf{g}_k = \left( b \mathbf{R}_{k-1} + x_k x_k^T \right) \mathbf{w}_k = b \mathbf{w}_{k-1} + x_k (y_k + \mu \delta_k)$$

$$= b \mathbf{R}_{k-1} \mathbf{w}_k + x_k x_k^T \mathbf{w}_k - b \mathbf{p}_{k-1} - x_k y_k - \mu \delta_k x_k$$  \quad (A.2)

Since the output of the linear combiner is the inner product of the weight and input vectors.

i.e., since $x_k^T \mathbf{w}_k = y_k$, Eq. (A.2) simplifies to:

$$\mathbf{g}_k = b \mathbf{R}_{k-1} \mathbf{w}_k - b \mathbf{p}_{k-1} - \mu \delta_k x_k$$  \quad (A.3)

The weight vector $\mathbf{w}_k$ at iteration $k$ is the sum of the weight vector $\mathbf{w}_{k-1}$ at iteration $k-1$ and the weight update at iteration $k-1$, $\Delta \mathbf{w}_{k-1}$, incorporating this in Eq. (A.3) yields:

$$\mathbf{g}_k = b \mathbf{R}_{k-1} (\mathbf{w}_{k-1} + \Delta \mathbf{w}_{k-1}) - b \mathbf{p}_{k-1} - \mu \delta_k x_k$$

$$= b (\mathbf{R}_{k-1} \mathbf{w}_{k-1} - \mathbf{p}_{k-1}) + \mathbf{R}_{k-1} \Delta \mathbf{w}_{k-1} - \mu \delta_k x_k$$

$$\mathbf{g}_k = b (\mathbf{g}_{k-1} + \mathbf{R}_{k-1} \Delta \mathbf{w}_{k-1}) - \mu \delta_k x_k$$  \quad (A.5)

where $\mathbf{g}_{k-1} + \mathbf{R}_{k-1} \Delta \mathbf{w}_{k-1}$ are quantities calculated at the previous iteration $k-1$, $x_k$ is the
input pattern vector currently presented to the layer in which the node is, and \( \delta_k \) is backpropagated through the hidden layers of the network. \( g_k \) can thus be recursively estimated using \( g_{k+1} \).

The next quantity of interest is \( \beta_k \) which is given by:

\[
\beta_k = g_k^T \frac{R_{k-1}d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}}
\]

which upon substitution of Eq. A-5 for \( g_k \) becomes:

\[
\beta_k = b(g_{k-1} + R_{k-1}\Delta w_{k-1})^T R_{k-1} \frac{d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}} - \mu \delta_k x_k^T R_{k-1} \frac{d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}}
\]  \(\text{(A-6)}\)

The direction \( d_k \) of the weight update is a combination of \( g_k \) and the previous direction \( d_{k-1} \):

\[
d_k = -g_{k-1} + \beta_k d_{k-1}
\]

which upon substituting Eq. A-5 and Eq A-6 becomes:

\[
d_k = -b(g_{k-1} + R_{k-1}\Delta w_{k-1}) + \mu \delta_k x_k + 
\]

\[
\left( b(g_{k-1} + R_{k-1}\Delta w_{k-1})^T R_{k-1} \frac{d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}} - \mu \delta_k x_k^T R_{k-1} \frac{d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}} \right) d_{k-1}
\]

\[
d_k = b \left( (g_{k-1} + R_{k-1}\Delta w_{k-1})^T R_{k-1} \frac{d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}} - (g_{k-1} + R_{k-1}\Delta w_{k-1}) \right) d_{k-1} + \mu \delta_k \left( x_k - x_k^T \frac{R_{k-1}d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}} \right)
\]  \(\text{(A-7)}\)
The step-size $\alpha_k$ is given by:

$$\alpha_k = -\frac{g_k^T d_k}{d_k^T R_k d_k}$$

Evaluating the numerator first yields:

$$g_k^T d_k = b[g_k - R_k \Delta w_{k-1} - \mu \delta_k x_k]^T[-b(g_k - R_k \Delta w_{k-1}) + \mu \delta_k x_k + \beta_k d_{k-1}]$$

$$= -b^2[g_k - R_k \Delta w_{k-1}]^T [g_k - R_k \Delta w_{k-1}] - \mu^2 \delta_k^2 x_k^T x_k + 2b\mu \delta_k x_k^T [g_k - R_k \Delta w_{k-1}]$$

$$- \mu \delta_k \beta_k x_k^T d_{k-1} + b\beta_k [g_k - R_k \Delta w_{k-1}]^T d_{k-1}$$

\[ (A-8) \]

Consider the last term in Eq. A-7:

$$b\beta_k [g_k - R_k \Delta w_{k-1}]^T d_{k-1} = b\beta_k [g_k - R_k \Delta w_{k-1}]^T d_{k-1}$$

$$= b\beta_k \left[ g_k - R_k \left( -\frac{g_k^T d_{k-1}}{d_{k-1}^T R_k d_{k-1}} \right) d_{k-1} \right]^T d_{k-1}$$

$$= b\beta_k \left[ g_k^T d_{k-1} + \left( -\frac{g_k^T d_{k-1}}{d_{k-1}^T R_k d_{k-1}} \right) \left( d_{k-1}^T R_k d_{k-1} \right) \right]$$

since the autocorrelation matrix $R_{k-1}$ is symmetric. The quadratic $d_{k-1}^T R_{k-1} d_{k-1}$ cancels out leaving:

$$b\beta_k [g_k - R_k \Delta w_{k-1}]^T d_{k-1} = b\beta_k \left[ g_k^T d_{k-1} - g_k^T R_{k-1} d_{k-1} \right] = 0$$

Hence Eq. A-7 reduces to:

$$g_k^T d_k = -b^2[g_k - R_k \Delta w_{k-1}]^T [g_k - R_k \Delta w_{k-1}] - \mu^2 \delta_k^2 x_k^T x_k$$

$$+ 2b\mu \delta_k x_k^T [g_k - R_k \Delta w_{k-1}] - \mu \delta_k \beta_k x_k^T d_{k-1}$$

\[ (A-9) \]

Substituting Eq. A-6 in the last term of Eq. A-9 yields:
\[
g_k^T d_k = -b^2 [g_{k-1} + R_{k-1} \Delta w_{k-1}]^T [g_{k-1} + R_{k-1} \Delta w_{k-1}] - \mu^2 \delta_k^2 x_k^T x_k + 2b \mu \delta_k x_k^T [g_{k-1} + R_{k-1} \Delta w_{k-1}]
\]

\[
- \mu \delta_k \left[ b(g_{k-1} + R_{k-1} \Delta w_{k-1})^T \frac{R_{k-1} d_{k-1}}{d_{k-1}^T R_{k-1} d_{k-1}} - \mu \delta_k x_k^T \frac{R_{k-1} d_{k-1}}{d_{k-1}^T R_{k-1} d_{k-1}} \right] x_k^T d_{k-1}
\]

(A-10)

Grouping like degrees of \( \delta_k \) yields:

\[
g_k^T d_k = \mu^2 \delta_k^2 \left( x_k^T \frac{R_{k-1} d_{k-1}}{d_{k-1}^T R_{k-1} d_{k-1}} x_k^T d_{k-1} - x_k^T x_k \right) + \mu b \delta_k x_k^T \left[ 2[g_{k-1} + R_{k-1} \Delta w_{k-1}] - [g_{k-1} + R_{k-1} \Delta w_{k-1}]^T \frac{R_{k-1} d_{k-1}}{d_{k-1}^T R_{k-1} d_{k-1}} d_{k-1} \right] - b^2 [g_{k-1} + R_{k-1} \Delta w_{k-1}]^T [g_{k-1} + R_{k-1} \Delta w_{k-1}] \]

(A-11)

Evaluating the quadratic in the denominator of \( a_k \), \( d_k^T R_k d_k \):

\[
d_k^T R_k d_k = b^2 [(g_{k-1} + R_{k-1} \Delta w_{k-1})^T \frac{R_{k-1} d_{k-1}}{d_{k-1}^T R_{k-1} d_{k-1}} d_{k-1} - g_{k-1} - R_{k-1} \Delta w_{k-1}]
\]

\[
+ \mu \delta_k (x_k - x_k^T \frac{R_{k-1} d_{k-1}}{d_{k-1}^T R_{k-1} d_{k-1}} d_{k-1}))^T R_k d_k
\]

Grouping like degrees of \( \delta_k \) yields:

\[
d_k^T R_k d_k = \eta_1 \delta_k^2 + \eta_1 \delta_k + \eta_0
\]

(A-12)

where:

\[
\eta_2 = \mu^2 \left[ x_k - x_k^T c_k d_{k-1} \right] R_k \left[ x_k - x_k^T c_k d_{k-1} \right]
\]
\[
\eta_i = b^T[h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1}]R_k[h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1}] + 2\mu[x_k - x_k^Tc_{k-1}d_{k-1}]^TR_k[h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1}]
\]

\[
\eta_0 = b[h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1} + \mu(x_k - x_k^Tc_{k-1}d_{k-1})]^TR_k[h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1} + \mu(x_k - x_k^Tc_{k-1}d_{k-1})]
\]

\[
c_{k-1} = \frac{R_{k-1}d_{k-1}}{d_{k-1}^TR_{k-1}d_{k-1}}
\]

\[
h_{k-1} = g_{k-1} + R_{k-1}\Delta w_{k-1}
\]

The weight update is given by: \(\Delta w_k = \alpha_k d_k\)

Substituting Eqs. A-7, A-11 and A-12 into the weight update above yields the weight change:

\[
\Delta w_k = \alpha_k d_k = \frac{n_3\delta_k^3 + n_2\delta_k^2 + n_1\delta_k + n_0}{\eta_2\delta_k^2 + \eta_1\delta_k + \eta_0}
\]  \(\text{(A-13)}\)

where

\[
n_3 = \mu^3(x_k^Tc_{k-1}d_{k-1} - x_k^Tc_{k-1}d_{k-1})[x_k - (x_k^Tc_{k-1}d_{k-1})]
\]

\[
n_2 = \mu^2b[x_k^T[2h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1}]][x_k - x_k^Tc_{k-1}d_{k-1}] + (x_k^Tc_{k-1}x_k^Td_{k-1} - x_k^Tx_k)\{h_{k-1}^Tc_{k-1}d_{k-1} - h_{k-1}\}
\]

\[
n_1 = \mu b^T[h_{k-1}^Tc_{k-1}[x_k^Tc_{k-1}d_{k-1} - x_k] + x_k^T[2h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1}]][h_{k-1}^Tc_{k-1}d_{k-1} - h_{k-1}]
\]

\[
n_0 = b^T[h_{k-1}^Tc_{k-1}[h_{k-1} - h_{k-1}^Tc_{k-1}d_{k-1}]]
\]
References:


