INCREMENTAL TRAINING ALGORITHMS FOR NONLINEAR NEURAL NETWORKS

By

ROGER L. SCHULTZ

Bachelor of Science
Oklahoma State University
Stillwater, Oklahoma
1986

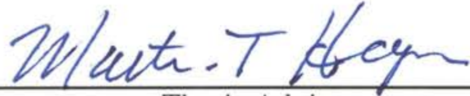
Master of Science
Oklahoma State University
Stillwater, Oklahoma
1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 2002

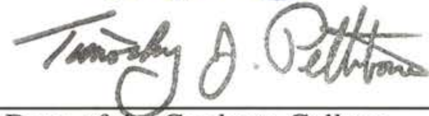
INCREMENTAL TRAINING ALGORITHMS FOR

NONLINEAR NEURAL NETWORKS

Thesis Approved:



Thesis Adviser



Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank Dr. Martin Hagan, for the generous help, encouragement and patience he has extended to me while teaching me so much. I would also like to thank my other committee members Dr. Carl Latino, Dr. Eduardo Misawa and Dr. George Sheets for their helpful suggestions and assistance. Many thanks also go to Meng Fun and Orlando De Jesus, two friends who have alway had good suggestions and a willingness to help.

I would also like to thank the management of Halliburton Energy Services for supporting and encouraging me in this work. I would especially like to thank Kent Beck, B.N. Murali, Harry Smith and Syed Hamid for their support.

I would like to express my deep appreciation to my wife, Annette, for her years of patience, loving support, and encouragement. Many thanks also go to my children Riley, Bailey, Aubrey and Wesley who have often had to endure my absence due to my pursuit of this endeavor. Finally, I wish to thank my parents for the encouraging example they have set, and for always supporting me in all that I do.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION 1

CHAPTER 2
BASIC BATCH OPTIMIZATION TECHNIQUES 7
Introduction 7
The Method of Steepest Descent 9
Steepest Descent Example (Batch Processing) 11
Newton's Method 16
Newton's Method Example 21
Gauss-Newton Method 25
Gauss-Newton Method Example 26
Levenberg-Marquardt Method 29
Levenberg-Marquardt Example 31
Linear Least-Squares 34
Nonlinear Least-Squares 36

CHAPTER 3
STANDARD INCREMENTAL OPTIMIZATION TECHNIQUES 40
Introduction 40
Method of Steepest Descent (On-line Processing) 41
Steepest Descent Example (On-line Processing) 42
Recursive Least-Squares 47
Recursive Gauss-Newton Method 52
Recursive Gauss-Newton Method Example 55
Extended Kalman Filter 58
Chapter Summary 67

CHAPTER 4
UNDERDETERMINED LINEARIZED LEAST-SQUARES TRAINING 68
Introduction 68
Overdetermined Linear Least Squares Solution (Review) 69
Underdetermined Linear Least Squares Solution 72
Nonlinear Least-Squares for the Overdetermined Case (Review) 74
Nonlinear Least-Squares for the Underdetermined Case 75
Neural Network Training Using the ULLS Method with Matrix Inversion 78
Recursive Underdetermined Linearized Least Squares (RULLS) Training Algorithm 80
Numerical Stability of the RULLS Algorithm 89
Chapter Summary 89

CHAPTER 5
INCREMENTAL LEVENBERG-MARQUARDT OPTIMIZATION 91
Introduction..91
Recursive Levenberg-Marquardt Equations 92
Recursive Levenberg-Marquardt Using Matrix Inversion Lemma 96
Strategies for adjusting 101
Determining 107
RLMMIL Simulation Results 110
Recursive Levenberg-Marquardt with Caley-Hamilton Approximation.114
RLMCH Method Simulation Results 118
Chapter Summary 120

CHAPTER 6
PERFORMANCE COMPARISON BETWEEN ULLS AND COMMON ON-LINE TRAINING METHODS...122
Introduction...122
Test Case 1: Non-Recurrent Neural Network Training Using Simulated Data...123
Test Case 2: Recurrent Neural Network Training Using Simulated Data...131
Test Case 3: Neural Network Training Using Experimental Voice Data135
Chapter Summary 137

CHAPTER 7
APPLICATION OF THE ULLS METHOD FOR DETECTING ROLLER-CONE DRILL BIT FAILURE 139
Introduction..139
Problem Description 140
ANNPA Method Experimental Verification 146
Downhole Tool and Warning System Description.................................151
Chapter Summary 154

CHAPTER 8
CONCLUSIONS AND RECOMMENDATIONS 155
Conclusions...155
Recommendations...159

REFERENCES...**162**

LIST OF TABLES

Table 1 Computational requirements for the RULLS algorithm 87

Table 2 Computational Requirements for Recursive LM Update (MIL) 99

Table 3 Computational Requirements for Full Recursive LM Update 100

Table 4 Comparison of One-Step and Incremental Matrix Inversion Lemma 106

Table 5 Training Method Performance Comparison 125

Table 6 Jacobian Terms Performance Comparison 128

Table 7 Jacobian Terms Performance Comparison (Recurrent Network, Fixed Weights) 133

LIST OF FIGURES

Figure 1 Example Nonlinear Neural Network 12

Figure 2 Squared-Error Performance for Batch Steepest Descent Example 13

Figure 3 Weight Trajectories for Batch Steepest Descent Example 13

Figure 4 Weight Trajectories for 2-Weight Steepest Descent Example 14

Figure 5 3-D Weight Trajectories for 2-Weight Steepest Descent Example 15

Figure 6 Weight Trajectories Contour Plot for 2-Weight Steepest Descent Example 15

Figure 7 Example Nonlinear Neural Network 21

Figure 8 Squared-Error Performance for Newton's Method Example 22

Figure 9 Weight Trajectories for Newton's Method Example 23

Figure 10 Weight Trajectories for 2-Weight Newton's Method Example 24

Figure 11 3-D Weight Trajectories for 3-Weight Newton's Method Example 24

Figure 12 Weight Trajectories Contour Plot for 2-Weight Newton's Method Example 25

Figure 13 Squared-Error Performance for Gauss-Newton Example 27

Figure 14 Weight Trajectories for Gauss-Newton Example 28

Figure 15 Weight Trajectories for 2-Weight Gauss-Newton Example 28

Figure 16 3-D Weight Trajectories for 3-Weight Gauss-Newton Example 29

Figure 17 Weight Trajectories Contour Plot for 2-Weight Gauss-Newton Example 29

Figure 18 Squared-Error Performance for Levenberg-Marquardt Example 31

Figure 19 Weight Trajectories for Levenberg-Marquardt Example 33

Figure 20 Weight Trajectories for 2-Weight Levenberg-Marquardt Example 33

Figure 21 3-D Weight Trajectories for 3-Weight Levenberg-Marquardt Example 34

Figure 22 Weight Trajectories Contour Plot for 2-Weight Levenberg-Marquardt Example 34

Figure 23 Example Nonlinear Neural Network 43

Figure 24 Squared-Error Performance for On-line Steepest Descent Example 44

Figure 25 Weight Trajectories for On-line Steepest Descent Example 44

Figure 26 Weight Trajectories for 2-Weight On-line Steepest Descent Example 45

Figure 27 3-D Weight Trajectories for 2-Weight On-line Steepest Descent Example 46

Figure 28 Weight Trajectories for 2-Weight On-line Steepest Descent Example 46

Figure 29 Squared-Error Performance for Recursive Gauss-Newton Example 56

Figure 30 Weight Trajectories for Recursive Gauss-Newton Example 56

Figure 31 Weight Trajectories for 2-Weight Recursive Gauss-Newton Example 57

Figure 32 3-D Weight Trajectories for 3-Weight Recursive Gauss-Newton Example 58

Figure 33 Weight Trajectories Contour Plot for 2-Weight RGN Example 58

Figure 34 Computation Requirement Comparison for ULLS and RULLS Algorithms 88

Figure 35 Computational Requirements for Single Recursive LM Update 100

Figure 36 Comparison of One-Step and Incremental Matrix Inversion Lemma 106

Figure 37 Example RLMMIL System Identification Training Problem 111

Figure 38 Squared-Error Performance for RLMMIL, RGN and Steepest Descent 112

Figure 39 Trajectory of during training (full case) 113

Figure 40 Squared-Error Performance for RLMMIL, RGN and Steepest Descent 114

Figure 41 Error Associated with Cayley-Hamilton Inverse Approximation 119

Figure 42 Three-Layer Feedforward Neural Network (Test Case 1) 123

Figure 43 Mean Squared-Error for Three Training Algorithms....................................124

Figure 44 Squared-Error vs. Flops for Three Training Methods....................................126

Figure 45 Effect of Jacobian Window Size On ULLS Performance (Non-Recurrent Network)..127

Figure 46 Slow Sinusoidal Parameter Change with ULLS Training 129

Figure 47 Rapid Sinusoidal Parameter Change with ULLS Training.............................130

Figure 48 Three-Layer Recurrent Neural Network (Test Case 2)..................................131

Figure 49 Effect of Jacobian Window Size On ULLS Performance (Recurrent Neural Network)..132

Figure 50 Effect of Jacobian Window Size On ULLS Performance..............................134

Figure 51 Non-contaminated Voice Signal 135

Figure 52 Contaminated Voice Signal...136

Figure 53 Speech Prediction Performance (ULLS, RGN and Steepest Descent) 137

Figure 54 Roller-Cone Drill Bit Bearing Failure Detection 142

Figure 55 Schematic of ANNPA Drill Bit Failure Detection Scheme 143

Figure 56 Adaptive Neural Network Predictor (ANNPA Method)................................143

Figure 57 Failure Indications (ANNPA) Method...145

Figure 58 Experimental Test Setup 147

Figure 59 Acceleration (no bearing damage) 148

Figure 60 Squared Prediction Error (no bearing damage)..149

Figure 61 Acceleration (moderate bearing damage) 149

Figure 62 Squared Prediction Error (moderate bearing damage)...................................150

Figure 63 Acceleration (heavy bearing damage)..151

Figure 64 Squared Prediction Error (heavy bearing damage) 151

Figure 65 Open Port Failure Indication 152

Figure 66 Downhole Tool Schematic. 153

Figure 67 Open-Close Signaling Operation 154

Chapter 1

INTRODUCTION

The objective of this research is to develop new optimization algorithms which are suitable for the adaptive real-time training of multilayer neural networks. For many years linear adaptive modeling has been used to solve numerous real-time filtering and controls problems [5]. In recent years the use of neural networks has become widespread in many areas of application. This includes the areas of adaptive filtering and control [24].

The use of neural networks in these types of applications gives rise to some unique difficulties both in real-time computational burden and in tracking performance. In filtering and controls applications it is common to use a tapped-delay [2] input structure. Traditionally, this input structure can create tens, to hundreds of inputs. In a linear filter structure (i.e. finite impulse response [2]) the number of parameters to be adjusted during adaptation is roughly equal to the number of inputs. If a linear filter structure is used, this number of parameters does not present a real-time implementation problem when using standard training algorithms such as LMS (least mean square) [5], or RLS (recursive least squares) [3]. It is sometimes desirable to replace a linear filter with a nonlinear multilayer neural network because of the potential for increased performance by the neural network. If a fully connected multilayer neural network is used with a typical tapped-delay input structure this can result in a relatively large number of weights which must be adjusted in

a real-time manner. Each neuron in the first layer of the network causes a 100% increase in the number of parameters as compared to a linear filter with the same number of inputs. This increase in the number of parameters can sharply increase the computational burden for implementation. The nonlinear, multilayer characteristics of a neural network require additional computation for the calculation of the function derivatives needed for training. The steepest descent algorithm can be used for online training of neural networks, but the performance does not match that of higher-order training methods which use approximate second-order derivatives in computing parameter updates. When higher-order training algorithms such as RGN (recursive Gauss-Newton) or EKF (extended Kalman filter) are used, the computational burden of these methods when applied to models having a large number of parameters, can be excessive. This is the motivation for this research. The desire is to develop new training algorithms which offer improved performance or reduced computational burden.

The contributive work presented in this thesis can generally be placed into two categories.

- Underdetermined linearized least squares optimization
- Recursive Levenberg-Marquardt algorithms

The objectives and a summarization of the work performed in each of these categories is provided below.

Underdetermined Linearized Least Squares

In this work an optimization algorithm based on the underdetermined least squares solution was developed. By using the underdetermined solution, it is possible to adjust a

larger number of parameters than there are samples available. The resulting computations involve smaller matrix manipulations and thus make the method more suitable for "on-line" implementation than most of the overdetermined solution methods. A fully recursive form of the algorithm was developed which does not require direct matrix inversion. A comparison was made between the ULLS algorithm and the recursive Gauss-Newton algorithm which showed that dramatically fewer floating point operations are required for the ULLS algorithm than the RGN algorithm while comparable performance can be achieved.

Recursive Levenberg-Marquardt Algorithm

A major portion of this research effort was the development of a recursive form of the Levenberg-Marquart algorithm. The excellent performance of the batch LM algorithm was a strong motivation for the effort expended to develop an incremental "on-line" version of the algorithm. Many different approaches were taken in the effort to develop a recursive LM algorithm.

Success was found in the application of the standard matrix inversion lemma used many times at each iteration, to account for the effect of the additional diagonal matrix terms required by the LM algorithm. Computational analysis of the algorithm showed that the recursive form requires less computation than any of the direct matrix inversion techniques except Gaussian Elimination. However, it has also been shown that if only a portion of the diagonal terms are adjusted at each time step, the new recursive LM algorithm maintains good performance but requires significantly less computation than using any of the direct matrix inversion methods. Further improvement can be made in the recursive algo-

rithm by developing a method of choosing which diagonal terms to adjust at a particular time step.

In order to discuss the new training methods to be presented here, it is necessary to examine some of the standard optimization methods used in training neural networks. These methods can be placed in two main categories. These are "batch" optimization algorithms and incremental optimization algorithms. Batch algorithms compute model parameters using all available data, all at once. This set of data is usually used repeatedly in "epochs", making a collective adjustment to the model parameters. Batch algorithms are presented in Chapter 2 and are demonstrated using a simple neural network training example. These algorithms are very important because they are the starting point for most incremental or recursive optimization algorithms.

In Chapter 3, incremental optimization algorithms are presented. Some are recursive, in that the current computation relies on a result from the previous time-step for the same computation for the present time-step. A common characteristic of all these algorithms is the computation of a parameter update at each time step These algorithms have largely been developed as incremental versions of corresponding batch algorithms. One exception to this is the Extended Kalman Filter which was developed directly from the standard Kalman filter which was originally formulated as a recursive algorithm. It is important to study these algorithms to gain insight into the mathematical development of recursive algorithms from batch forms.

During the review of the least squares algorithm, the concept of using the underdetermined [4] least squares solution for training neural networks was considered. This is the

subject of Chapter 4. A complete development of the underdetermined least squares solution for linear functions is given. This is followed by the development of an extension for use with nonlinear functions. This method is then further developed into a neural network training algorithm. The use of the underdetermined solution has the advantage of making it possible to update more parameters than there are samples available. This training algorithm is the primary focus of this research.

One of the batch optimization methods covered in Chapter 2 is the Levenberg-Marquardt algorithm. This algorithm has been used very effectively in training multilayer neural networks. The success of this algorithm as a method for training neural networks is the motivation for the new work presented in Chapter 5. In Chapter 5 two different approaches for developing a recursive form of the Levenberg-Marquardt algorithm are presented. This algorithm was developed as a modification to the recursive Gauss-Newton algorithm. The mathematical form of the batch algorithm does not lend itself well to the development of a recursive algorithm. One of the new methods presented in Chapter 5 relies on an inverse approximation and the other uses the matrix inversion lemma [3] to compute a required inverse.

In Chapter 6 performance comparisons are made between the new ULLS (underdetermined linearized least squares) training method and two common training methods. The two methods selected for comparison are the steepest descent method and the RGN (recursive Gauss-Newton) method. Comparisons are made based on squared-error performance and computational complexity. Three different test problems are used in the comparison. The test problems include a three-layer multilayer feedforward network, a three-layer mul-

tilayer recurrent network using dynamic backpropagation and an example in which the various methods are used to restore a noisy speech signal. Both qualitative and quantitative comparisons are made.

In Chapter 7 a real-world application in which the ULLS algorithm is used is presented. In this application a neural network is used in the detection of failures in oilwell drill bit bearings. A description of the method is given as well as experimental results using real test data.

In the final chapter, conclusions based on the work presented here and recommendations for future work are provided.

Chapter 2

BASIC BATCH OPTIMIZATION TECHNIQUES

Introduction

In all function minimization methods the task is to find the appropriate unknown function parameters Θ which minimize the function. In the training of neural networks, the function to be minimized is a cost function which usually contains one or more squared-error terms. The error is the difference between a desired set of network outputs and the actual network outputs. The "function parameters" to be optimized are the neural network weights, which determine the network output.

Methods for numerical minimization involve iteratively updating the estimates of the parameters which minimize a function $F(\Theta_n)$. In searching for the optimum Θ, a new estimate may be represented as a combination of the previous estimate summed with a incremental change as in

$$\Theta_{n+1} = \Theta_n + \Delta\Theta_{n_{OPT}} \tag{1}$$

The problem is to find the series of incremental adjustments $\Delta\Theta_n$ which will lead to the function minimum. The basic parameter adjustment equation (Eq. (1)) is the same for all numerical minimization techniques. The difference in the various methods lies in the method of choosing $\Delta\Theta_n$. The incremental adjustment $\Delta\Theta_n$ is determined by a search di-

rection and a step size which controls the size of the parameter adjustment in the search direction, as in

$$\Theta_{n+1} = \Theta_n + \alpha \boldsymbol{p}_n \tag{2}$$

In Eq. (2), α is a positive constant step size, and the vector $\boldsymbol{p}_n$ is the search direction. The choice of the search direction is typically based on information obtained from previous iterations. The step size is often adjusted in response to some convergence criteria, but is sometimes fixed, depending on the specific method. All the numerical optimization methods which will be discussed in this chapter can be placed in one of two categories. These categories are:

1. Methods which utilize function gradient (first-order derivative) values only.

2. Methods which utilize function gradient and Hessian (second-order derivative).

Methods of the first type are in general less computationally intensive than methods which use higher-order terms. Methods which use only first-order derivatives also tend to be more readily implemented recursively. These methods also tend to converge less quickly than the methods which use both gradient and Hessian values [1]. In this chapter four methods which use only first-order derivatives will be presented. One of these first-order methods is the basic steepest descent method. Two other methods which use only first-order derivatives but approximate the higher-order Newton's method will also be discussed. These are the Gauss-Newton method and the Levenberg-Marquardt method. Newton's method will also be presented. A simple illustrative example showing the parameter trajectories for each of the methods will be presented as well.

The Method of Steepest Descent

One common and widely used numerical minimization method is known as the steepest-descent or gradient-descent method [2]. This method uses only first-order derivatives of the function to be minimized. As described earlier, an equation of the form

$$\Theta_{n+1} = \Theta_n + \alpha \mathbf{p}_n \tag{3}$$

is used to iteratively search for the minimum point of the function to be minimized. In Eq. (3) α is a small positive number so the problem is to determine a search direction $\mathbf{p}_n$ which will cause $F(\Theta_{n+1})$ to be smaller than $F(\Theta_n)$. To examine this problem it is helpful to look at the first-order Taylor series expansion of the function to be minimized. The first-order Taylor series expansion of $F(\Theta_{n+1})$ can be written as:

$$F(\Theta_{n+1}) = F(\Theta_n + \alpha \boldsymbol{p}_n) \approx F(\Theta_n) + \alpha \nabla F(\Theta)^T\big|_{\Theta = \Theta_n} \mathbf{p}_n \tag{4}$$

Where $\nabla F(\Theta)$ is the gradient:

$$\nabla F(\Theta) = \begin{bmatrix} \frac{\partial}{\partial \Theta_1} F(\Theta) \\ \frac{\partial}{\partial \Theta_2} F(\Theta) \\ \vdots \\ \frac{\partial}{\partial \Theta_M} F(\Theta) \end{bmatrix} \tag{5}$$

It is obvious upon examination of Eq. (4) that the inner product $\nabla F(\Theta)^T\big|_{\Theta = \Theta_n} \mathbf{p}_n$ which appears in Eq. (4) must be negative in order for the new function value to be less than the previous function value. It is desirable to find the search direction which will produce the

biggest decrease in F. When the search direction is precisely the negative of the gradient the inner-product is most negative. Therefore the steepest decent search direction is the negative of the gradient. With this notion we can modify Eq. (3) in the following way:

$$\Theta_{n+1} = \Theta_n - \alpha[\nabla F(\Theta)|_{\Theta = \Theta_n}] \tag{6}$$

Eq. (6) represents the steepest decent method.

For estimating the parameters of an unknown function, we often use the sum-of-the-squared-errors performance function:

$$F(\Theta) = \sum_{j=1}^{N} e_j^{\,2}(\Theta) = \mathbf{e}^T(\Theta)\mathbf{e}(\Theta) \tag{7}$$

where:

$$\mathbf{e}(\Theta) = [\mathbf{t} - \mathbf{a}(\Theta)] \tag{8}$$

In Eq. (7), $\mathbf{e}(\Theta)$ represents the error, $\mathbf{t}$ represents the target, and $\mathbf{a}(\Theta)$ represents the approximation function output. This problem is known as the classic least-squares problem [1], which can be linear or nonlinear depending on the type of approximation function used.

To minimize Eq. (7), we adjust the parameters of the function to minimize the difference between the target outputs for a given set of inputs, and the function outputs for the same set of inputs. This minimization can be accomplished by applying the gradient method. The gradient of Eq. (7) can be expressed as:

$$\nabla F(\Theta) = \nabla(e_1^2(\Theta) + e_2^2(\Theta) + e_3^2(\Theta) + \ldots + e_N^2(\Theta)) \tag{9}$$

or,

$$\nabla F(\Theta) = 2\begin{bmatrix} e_1(\Theta)\frac{\partial e_1(\Theta)}{\partial\Theta_1} + e_2(\Theta)\frac{\partial e_2(\Theta)}{\partial\Theta_1} + \ldots + e_N(\Theta)\frac{\partial e_N(\Theta)}{\partial\Theta_1} \\ e_1(\Theta)\frac{\partial e_1(\Theta)}{\partial\Theta_2} + e_2(\Theta)\frac{\partial e_2(\Theta)}{\partial\Theta_2} + \ldots + e_N(\Theta)\frac{\partial e_N(\Theta)}{\partial\Theta_2} \\ \vdots \\ e_1(\Theta)\frac{\partial e_1(\Theta)}{\partial\Theta_M} + e_2(\Theta)\frac{\partial e_2(\Theta)}{\partial\Theta_M} + \ldots + e_N(\Theta)\frac{\partial e_N(\Theta)}{\partial\Theta_M} \end{bmatrix} \tag{10}$$

The process of training a multilayer nonlinear network using steepest descent can best be illustrated by way of an example which will be given in the following section.

Steepest Descent Example (Batch Processing)

Suppose we have the network that is shown in Figure 1. For this example the network has 1 input, 1 nonlinear layer, and a linear output layer which produces a single output. The nonlinear activation functions are log-sigmoid functions which are described by equations (11) and (12) below.

$$f(n) = \frac{1}{1+e^{-n}} \tag{11}$$

$$\frac{\mathrm{d}}{\mathrm{d}n}f(n) = \frac{1}{1+e^{-n}}\left[1 - \frac{1}{1+e^{-n}}\right] \tag{12}$$

There are three parameters, w_1, w_2 and w_3, which must be adjusted to appropriate values using the steepest descent method.

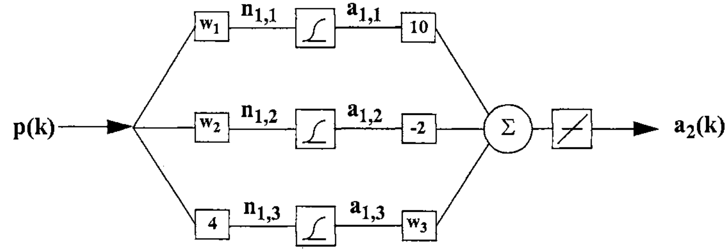


Figure 1 Example Nonlinear Neural Network

Test data were generated by setting and fixing all the network weights to known values, then applying a set of inputs and recording the corresponding outputs. The steepest descent method was then applied to the same network structure with the three unknown weights initialized to small random numbers.

This example will demonstrate how steepest-descent can be used to determine the "unknown" network weights which will provide an input/output relationship which matches the test data. It is possible for there to be multiple solutions to the problem, so it should be expected that the network weights will not always converge to exactly the same values as those contained in the network used to generate the test data.

In the context of function minimization, the idea here is to minimize the sum of the squared errors function given by Eq. (7) and repeated here for convenience.

$$F(\Theta) = \sum_{j=1}^{N} e_j^2(\Theta) = e^T(\Theta)e(\Theta) \tag{13}$$

There are 3 unknown network weights (represented by Θ) which must be adjusted to minimize Eq. (13). For this example a set of 100 input/output pairs were used in training the network. This data set was processed using the batch steepest descent algorithm. The three

unknown weights were initialized to small random numbers before training was performed. Figure 2 shows a plot of the sum of the squared errors as a function of epoch number. The trajectories of the unknown weights are shown in Figure 3. In this plot it can be seen that after 100 epochs of training the weights have essentially converged

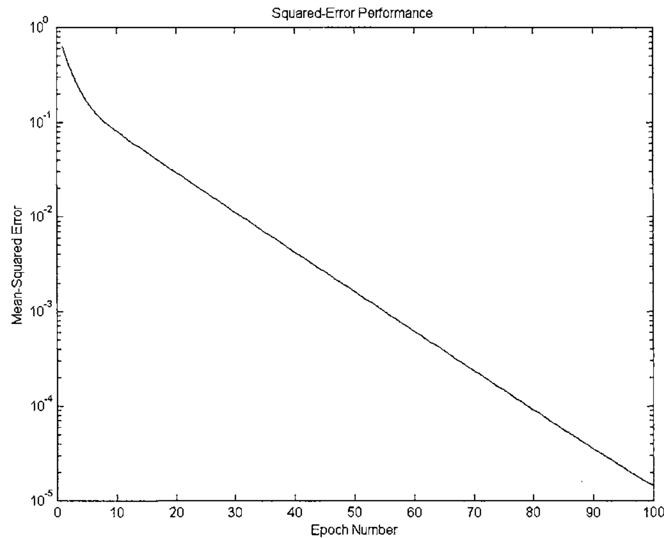


Figure 2 Squared-Error Performance for Batch Steepest Descent Example

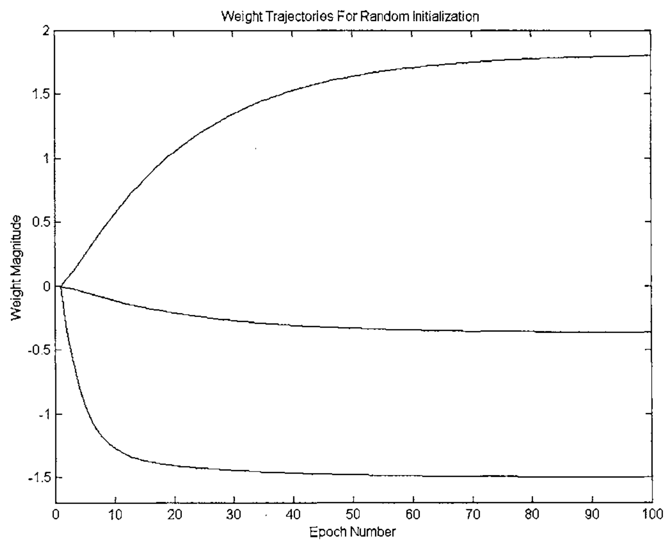


Figure 3 Weight Trajectories for Batch Steepest Descent Example

In the next part of this example, all but two of the network weights were fixed, and these two weights were adjusted using the steepest descent method. Having only two weights to adjust allows for graphical representation of the training process. The network weights were set to 2.5, 3.0, and -1.5 to generate the training data. Training data was then generated as described previously. The network weights were then initialized to -4.5, 3.0 and -4.5 and weights w_1 and w_3 were adjusted using the steepest descent method, while w_2 was held fixed at its original value. In this case we should expect to see the weights return to the values used in generating the training data. Indeed, when the training was performed, the weights converged on the original values, as shown in Figure 4. Three-dimensional and contour plots of the weights traversing the squared-error surface are shown in Figure 5 and Figure 6.

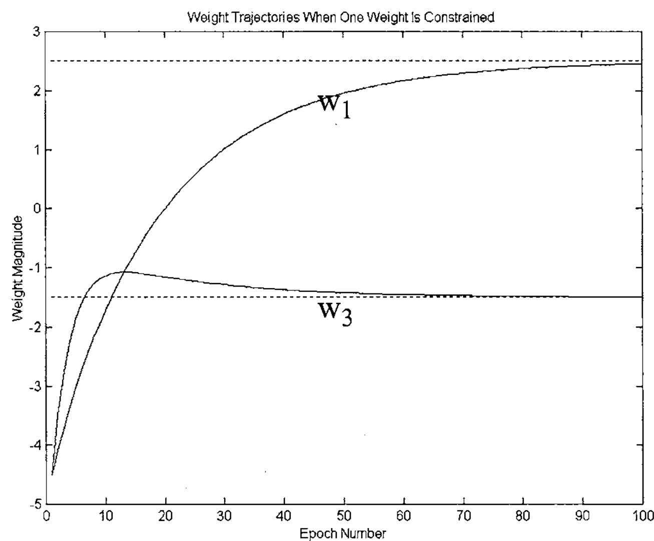


Figure 4 Weight Trajectories for 2-Weight Steepest Descent Example

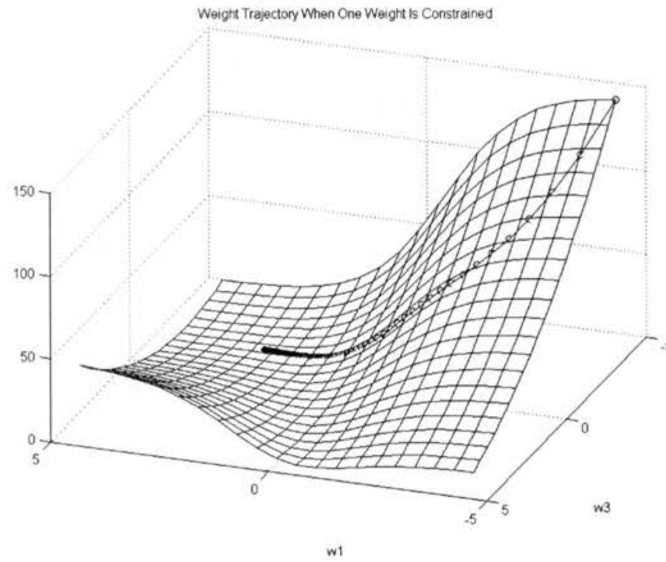


Figure 5 3-D Weight Trajectories for 2-Weight Steepest Descent Example

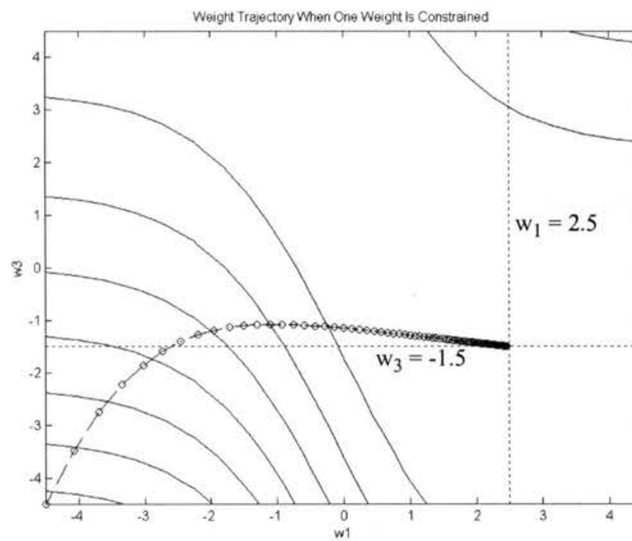


Figure 6 Weight Trajectories Contour Plot for 2-Weight Steepest Descent Example

Figure 5 illustrates how the steepest-descent trajectory must stay “close” to the error surface in order for the parameter adjustments to reduce the error. This limits the size of the weight adjustment which can be made when using this method. In Figure 6 we can see that

the weight trajectory progresses rapidly in the early steps, but then becomes relatively slow near the minimum point. This makes sense because the gradient becomes very small near the minimum. The algorithm relies on multiplying a small learning rate times the gradient to compute parameter adjustments making adjustments, close the minimum point, very small.

Newton's Method

Another approach to function minimization is known as Newton's Method [2]. In Newton's method, a second order Taylor series expansion of the function to be minimized is used to approximate the function. This quadratic approximation is then minimized. The second order Taylor series expansion of a function $F(\Theta_{n+1})$ can be written as:

$$F(\Theta_{n+1}) = F(\Theta_n + \Delta\Theta_n) \approx F(\Theta_n) + \nabla F(\Theta)^T\big|_{\Theta = \Theta_n}\Delta\Theta_n + \frac{1}{2}\Delta\Theta_n^T\nabla^2 F(\Theta)\big|_{\Theta = \Theta_n}\Delta\Theta_n \quad (14)$$

The second-order partial derivative of the function with respect to the unknown function parameters which appears in Eq. (14) can be expanded as:

$$\nabla^2 F(\Theta) = \begin{bmatrix} \frac{\partial^2}{\partial\Theta_1^2}F(\Theta) & \frac{\partial^2}{\partial\Theta_1\partial\Theta_2}F(\Theta) & \cdots & \frac{\partial^2}{\partial\Theta_1\partial\Theta_M}F(\Theta) \\ \frac{\partial^2}{\partial\Theta_2\partial\Theta_1}F(\Theta) & \frac{\partial^2}{\partial\Theta_2^2}F(\Theta) & \cdots & \frac{\partial^2}{\partial\Theta_2\partial\Theta_M}F(\Theta) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2}{\partial\Theta_M\partial\Theta_1}F(\Theta) & \frac{\partial^2}{\partial\Theta_M\partial\Theta_2}F(\Theta) & \cdots & \frac{\partial^2}{\partial\Theta_M^2}F(\Theta) \end{bmatrix} \quad (15)$$

Taking the gradient of Eq. (14) and setting the result equal to zero results in the following equation:

$$\nabla F(\Theta)^T\big|_{\Theta = \Theta_n} + \nabla^2 F(\Theta)\big|_{\Theta = \Theta_n} \Delta\Theta_n = \mathbf{0} \tag{16}$$

Eq. (16) can be rearranged in the following way to provide a means for calculating the optimum parameter adjustment to minimize the quadratic approximation to the function $F(\Theta)$.

$$\Delta\Theta_{n_{OPT}} = [-\nabla^2 F(\Theta)\big|_{\Theta = \Theta_n}]^{-1} \nabla F(\Theta)^T\big|_{\Theta = \Theta_n} \tag{17}$$

Of course, in the general case, the function $F(\Theta)$ will not be quadratic, and will not be minimized in one step. The basic idea of Newton's method is to use the second order Taylor series approximation to the function at each iteration until a minimum is reached. This process can be described by the following equation:

$$\Theta_{n+1} = \Theta_n + \Delta\Theta_{n_{OPT}} \tag{18}$$

For neural network training, the function to be minimized is the sum of squared-errors function:

$$F(\Theta) = \sum_{j=1}^{N} e_j^2(\Theta) = \mathbf{e}^T(\Theta)\mathbf{e}(\Theta) \tag{19}$$

where: $$\mathbf{e}(\Theta) = [\mathbf{t} - \mathbf{a}(\Theta)] \tag{20}$$

In Eq. (20) $\mathbf{e}(\Theta)$ represents the vector of function errors or vector of differences between the target output vector $\mathbf{t}$, and the network output vector $\mathbf{a}(\Theta)$.

This minimization of Eq. (19) may be accomplished using Newton's method. Eq. (19) is a function of the network parameters, so it can be minimized with respect to the network parameters. In order to apply Newton's method to Eq. (19) we must use Eq. (17) and

Eq. (18). The gradient of Eq. (19) with respect to the unknown parameters can be expressed as:

$$\nabla F(\Theta) = \nabla(e_1^2(\Theta) + e_2^2(\Theta) + e_3^2(\Theta) + \ldots + e_N^2(\Theta)) \tag{21}$$

or,

$$\nabla F(\Theta) = 2\begin{bmatrix} e_1(\Theta)\frac{\partial e_1(\Theta)}{\partial \Theta_1} + e_2(\Theta)\frac{\partial e_2(\Theta)}{\partial \Theta_1} + \ldots + e_N(\Theta)\frac{\partial e_N(\Theta)}{\partial \Theta_1} \\ e_1(\Theta)\frac{\partial e_1(\Theta)}{\partial \Theta_2} + e_2(\Theta)\frac{\partial e_2(\Theta)}{\partial \Theta_2} + \ldots + e_N(\Theta)\frac{\partial e_N(\Theta)}{\partial \Theta_2} \\ \vdots \\ e_1(\Theta)\frac{\partial e_1(\Theta)}{\partial \Theta_M} + e_2(\Theta)\frac{\partial e_2(\Theta)}{\partial \Theta_M} + \ldots + e_N(\Theta)\frac{\partial e_N(\Theta)}{\partial \Theta_M} \end{bmatrix} \tag{22}$$

Eq. (22) can be rewritten as:

$$\nabla F(\Theta) = \begin{bmatrix} \frac{\partial e_1(\Theta)}{\partial \Theta_1} & \frac{\partial e_2(\Theta)}{\partial \Theta_1} & \cdots & \frac{\partial e_N(\Theta)}{\partial \Theta_1} \\ \frac{\partial e_1(\Theta)}{\partial \Theta_2} & \frac{\partial e_2(\Theta)}{\partial \Theta_2} & \cdots & \frac{\partial e_N(\Theta)}{\partial \Theta_2} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_1(\Theta)}{\partial \Theta_M} & \frac{\partial e_2(\Theta)}{\partial \Theta_M} & \cdots & \frac{\partial e_N(\Theta)}{\partial \Theta_M} \end{bmatrix} \begin{bmatrix} e_1(\Theta) \\ e_2(\Theta) \\ \vdots \\ e_N(\Theta) \end{bmatrix} \tag{23}$$

or

$$\nabla F(\Theta) = 2\mathbf{J}^T(\Theta)\boldsymbol{e}(\Theta) \;, \tag{24}$$

where

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e_1(\Theta)}{\partial \Theta_1} & \frac{\partial e_1(\Theta)}{\partial \Theta_2} & \cdots & \frac{\partial e_1(\Theta)}{\partial \Theta_M} \\ \frac{\partial e_2(\Theta)}{\partial \Theta_1} & \frac{\partial e_2(\Theta)}{\partial \Theta_2} & \cdots & \frac{\partial e_2(\Theta)}{\partial \Theta_M} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_N(\Theta)}{\partial \Theta_1} & \frac{\partial e_N(\Theta)}{\partial \Theta_2} & \cdots & \frac{\partial e_N(\Theta)}{\partial \Theta_M} \end{bmatrix} \tag{25}$$

J is commonly known as the Jacobian matrix [2]. Notice that the elements of the Jacobian matrix contain the first derivatives of the error with respect to each of the unknown function parameters.

In order to use Eq. (17), the Hessian matrix $\nabla^2 F(\Theta)$ must be calculated. It can be written as:

$$\nabla^2 F(\Theta) = 2\begin{bmatrix} \left(\frac{\partial e_1(\Theta)}{\partial \Theta_1}\frac{\partial e_1(\Theta)}{\partial \Theta_1} + \ldots\right) & \cdots & \left(\ldots + \frac{\partial e_N(\Theta)}{\partial \Theta_M}\frac{\partial e_N(\Theta)}{\partial \Theta_1}\right) \\ \left(\frac{\partial e_1(\Theta)}{\partial \Theta_1}\frac{\partial e_1(\Theta)}{\partial \Theta_2} + \ldots\right) & \cdots & \left(\ldots + \frac{\partial e_N(\Theta)}{\partial \Theta_M}\frac{\partial e_N(\Theta)}{\partial \Theta_2}\right) \\ \vdots & & \vdots \\ \left(\frac{\partial e_1(\Theta)}{\partial \Theta_1}\frac{\partial e_1(\Theta)}{\partial \Theta_M} + \ldots\right) & \cdots & \left(\ldots + \frac{\partial e_N(\Theta)}{\partial \Theta_M}\frac{\partial e_N(\Theta)}{\partial \Theta_M}\right) \end{bmatrix}$$

$$+ 2\begin{bmatrix} \left(e_1(\Theta)\frac{\partial^2 e_1(\Theta)}{\partial \Theta_1 \partial \Theta_1} + \ldots\right) & \cdots & \left(\ldots + e_N(\Theta)\frac{\partial^2 e_N(\Theta)}{\partial \Theta_1 \partial \Theta_M}\right) \\ \left(e_1(\Theta)\frac{\partial^2 e_1(\Theta)}{\partial \Theta_2 \partial \Theta_1} + \ldots\right) & \cdots & \left(\ldots + e_N(\Theta)\frac{\partial^2 e_N(\Theta)}{\partial \Theta_2 \partial \Theta_M}\right) \\ \vdots & & \vdots \\ \left(e_1(\Theta)\frac{\partial^2 e_1(\Theta)}{\partial \Theta_M \partial \Theta_1} + \ldots\right) & \cdots & \left(\ldots + e_N(\Theta)\frac{\partial^2 e_N(\Theta)}{\partial \Theta_M \partial \Theta_M}\right) \end{bmatrix} \tag{26}$$

Eq. (26) can be conveniently re-written in a more compact form as:

$$\nabla^2 F(\Theta) = 2\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + 2\mathbf{U}(\Theta) \tag{27}$$

The matrix **J** is the Jacobian matrix as defined before, and the matrix $\boldsymbol{U}$ is the matrix containing the higher-order terms of $\nabla^2 F(\Theta)$. Rewriting Eq. (18), Eq. (17), Eq. (27) and Eq. (24) we can summarize Newton's method for the squared error function error function as:

$$\Theta_{n+1} = \Theta_n + \Delta\Theta_{n_{OPT}} \tag{28}$$

$$\Delta\Theta_{n_{OPT}} = -[\nabla^2 F(\Theta)|_{\Theta = \Theta_n}]^{-1} \nabla F(\Theta)^T|_{\Theta = \Theta_n} \tag{29}$$

$$\nabla^2 F(\Theta) = 2\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + 2\mathbf{U}(\Theta) \tag{30}$$

$$\nabla F(\Theta) = 2\mathbf{J}^T(\Theta)e(\Theta) \tag{31}$$

or,

$$\Theta_{n+1} = \Theta_n - [\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + \mathbf{U}(\Theta)]^{-1}\mathbf{J}^T(\Theta)e(\Theta) \tag{32}$$

with the matrices **J** and **U** defined previously in Eq. (26) and Eq. (27).

Newton's method usually performs much better than steepest descent near the function minimum, where the gradient may be very shallow [1]. This increase in performance of Newton's method is costly in terms of computational intensity. It takes many more floating operations for one iteration of Newton's method than is required for a single steepest descent iteration. The calculation of the terms needed to construct the Hessian matrix defined in Eq. (26) and the computation or the inverse of the Hessian account for the great increase in computational burden associated with Newton's method. The second-order derivatives contained in the $\mathbf{U}(\Theta)$ matrix in Eq. (27) are particularly troublesome, because

these terms cannot be expressed as combinations of first-order derivatives, as is the case with the terms of $\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)$ in the same equation. A mutilayer nonlinear network training example using Newton's method follows.

Newton's Method Example

Suppose we have the same example network that was trained previously using steepest descent. The network is shown again for convenience in Figure 7. For this example the network has 1 input, 1 nonlinear layer, and a linear output layer which produces a single output.

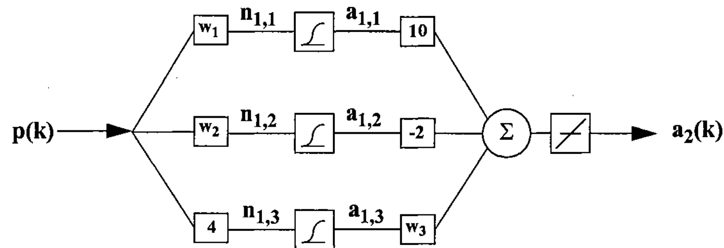


Figure 7 Example Nonlinear Neural Network

To perform Newton's method, Eq. (32) was used. This process was performed for 20 epochs. Figure 8 shows a plot of the sum of the squared errors versus epoch number. The three unknown weights were initialized to small random numbers before training was performed.

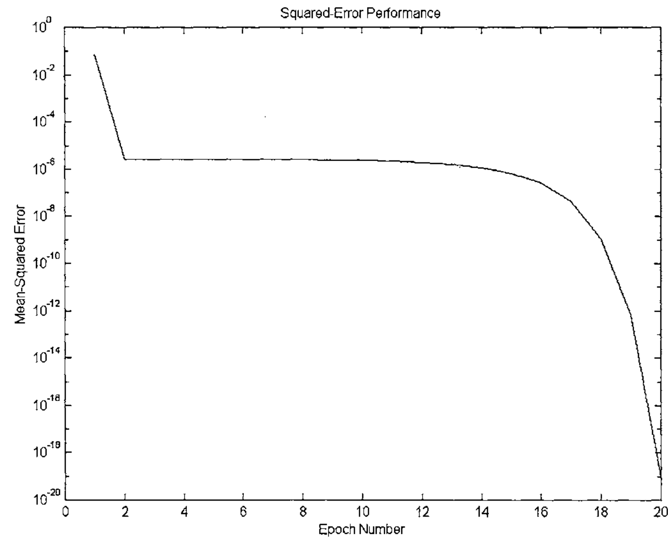


Figure 8 Squared-Error Performance for Newton's Method Example

The trajectories of the unknown weights as training progresses are shown in Figure 9. In this plot is can be seen that after 20 epochs of training the weights have converged. Notice how much faster Newton's method converges for this example than steepest descent did.

In the next part of this example, all but two of the network weights were fixed, and the two weights adjusted using Newton's method. The network weights were set to 2.5, 3.0, and -1.5 to generate the training data. The network weights were then initialized to -4.5, 3.0 and -4.5, and weights w_1 and w_3 adjusted using the Newton's method, while w_2 was held fixed at its original value. In this case we should expect to see the weights return to the values used in generating the training data. Indeed when the training was performed, the weights converged on the original values as shown in Figure 10. Again notice how fast the weights converged. Three-dimensional and contour plots of the weights traversing the

squared-error surface are shown in Figure 11 and Figure 12. Notice in Figure 11 that the trajectory "cuts" through the error surface and jumps to a new location without being constrained to the error surface as is the case with steepest descent. It can be seen in Figure 12 how few steps are needed with Newton's to method approach the optimum weights for minimizing the error.

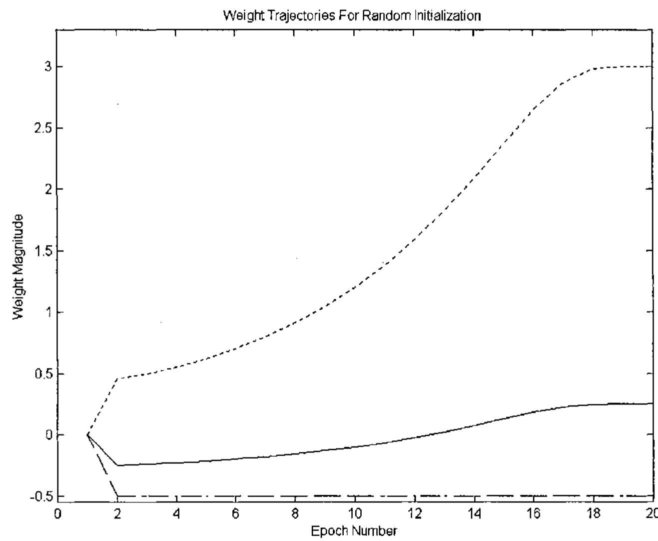


Figure 9 Weight Trajectories for Newton's Method Example

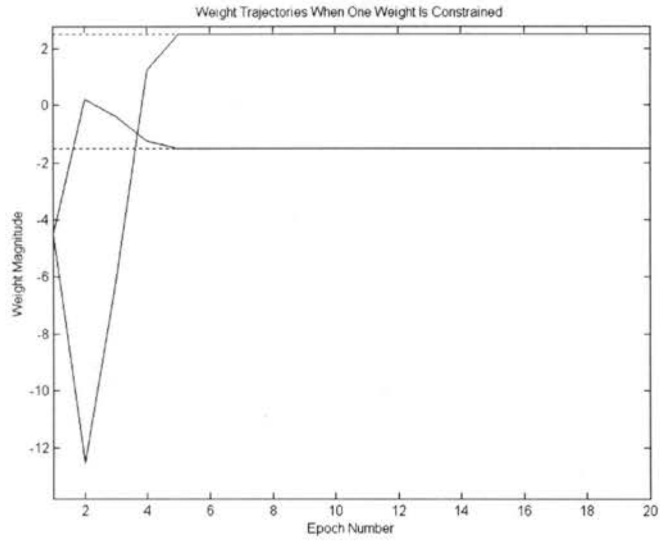


Figure 10 Weight Trajectories for 2-Weight Newton's Method Example

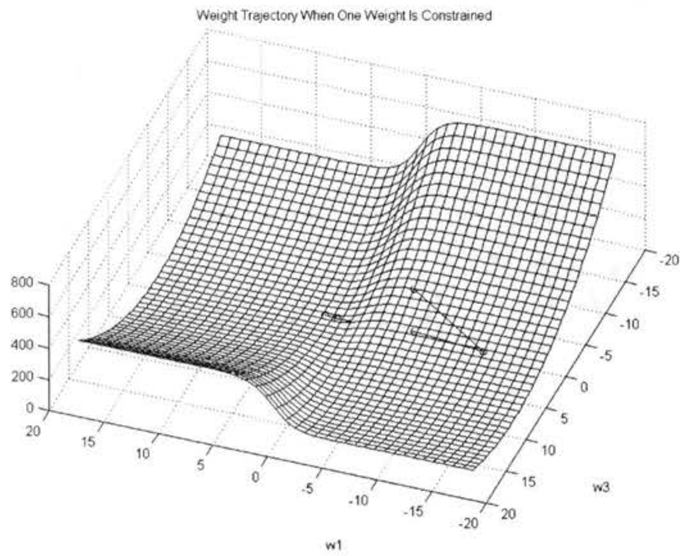


Figure 11 3-D Weight Trajectories for 3-Weight Newton's Method Example

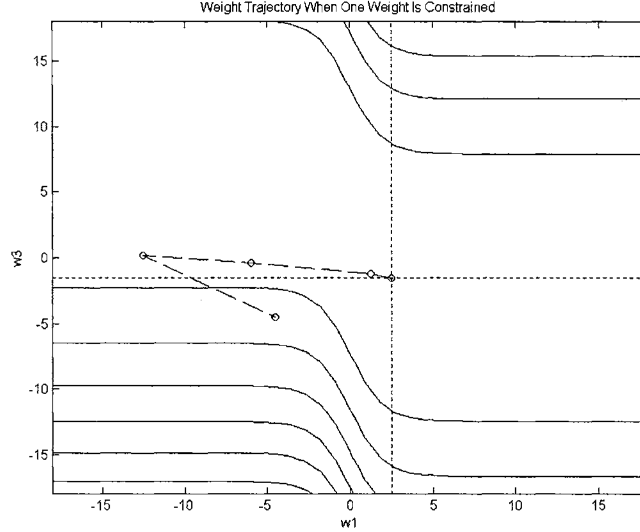


Figure 12 Weight Trajectories Contour Plot for 2-Weight Newton's Method Example

Gauss-Newton Method

The Gauss-Newton method [2] is one of a group of minimization methods which are based on Newton's method, but have been modified to reduce the computation burden associated with calculating the Hessian matrix (Eq. (26), Eq. (27)). Recall that for Newton's method the parameter adjustments are:

$$\Delta\Theta_{n_{OPT}} = -[\nabla^2 F(\Theta)|_{\Theta = \Theta_n}]^{-1} \nabla F(\Theta)^T|_{\Theta = \Theta_n} \tag{33}$$

where:

$$\nabla^2 F(\Theta) = 2\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + 2\mathbf{U}(\Theta) \tag{34}$$

The elements of the Jacobian matrix $\mathbf{J}$ contain only first derivatives. These terms can be calculated relatively easily. On the other hand, the elements of $\mathbf{U}$ include second derivatives, which add a new level of complexity and computational burden to the Hessian matrix determination. If the higher-order terms contained in $\mathbf{U}$ are considered to be neg-

ligibly small, then $\nabla^2 F(\Theta)$ can be approximated as shown in Eq. (35) by simply omitting these terms.

$$\nabla^2 F(\Theta) \approx 2\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) \tag{35}$$

This modification yields the Gauss-Newton method, which only requires the computation of first-order derivatives, greatly reducing the computational requirements of Newton's method. The iterative Gauss-Newton algorithm can be summarized as:

$$\Theta_{n+1} = \Theta_n + \Delta\Theta_{n_{OPT}} \tag{36}$$

$$\Delta\Theta_{n_{OPT}} = -[\nabla^2 F(\Theta)|_{\Theta = \Theta_n}]^{-1} \nabla F(\Theta)^T|_{\Theta = \Theta_n} \tag{37}$$

$$\nabla^2 F(\Theta) \approx 2\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) \tag{38}$$

$$\nabla F(\Theta) = 2\mathbf{J}^T(\Theta)e(\Theta) \tag{39}$$

or,

$$\Theta_{n+1} = \Theta_n - [\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)]^{-1}\mathbf{J}^T(\Theta)e(\Theta) \tag{40}$$

Gauss-Newton Method Example

The Gauss-Newton method was applied to the same example network shown in Figure 7 that was trained using steepest descent and Newton's method in the previous examples. 20 iterations of the algorithm were performed. Figure 13 shows a plot of the sum of the squared errors versus epoch number. The three unknown weights were initialized to small random numbers before training was performed.

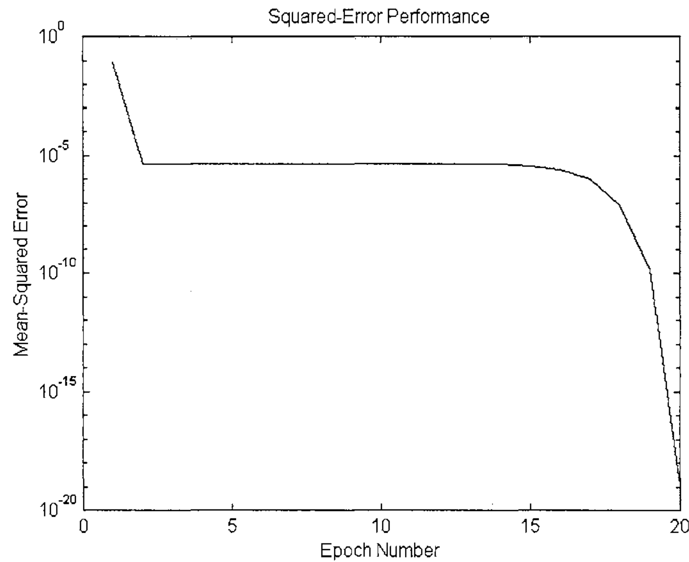


Figure 13 Squared-Error Performance for Gauss-Newton Example

The trajectories of the unknown weights as training progresses are shown in Figure 14. In this plot is can be seen that after 20 epochs of training the weights have converged. Notice how the Gauss-Newton method converges with approximately the same speed as Newton's method, but much faster than steepest descent.

To further illustrate the Gauss-Newton method all but two of the network weights were fixed, and these two weights were adjusted using the method. The network weights were set to 2.5, 3.0, and -1.5 for generating training data. The network weights were then initialized to -4.5, 3.0 and -4.5, and weights w_1 and w_3 were adjusted using the Gauss-Newton method, while w_2 was held fixed at it's original value. As training progressed, the weights converged on the original values as shown in Figure 15. The trajectory for the weights are similar to those for Newton' method. Three-dimensional and contour plots of the weights traversing the squared-error surface are shown in Figure 16 and Figure 17. Notice in Figure 16 that the trajectory is similar to Newton's method.

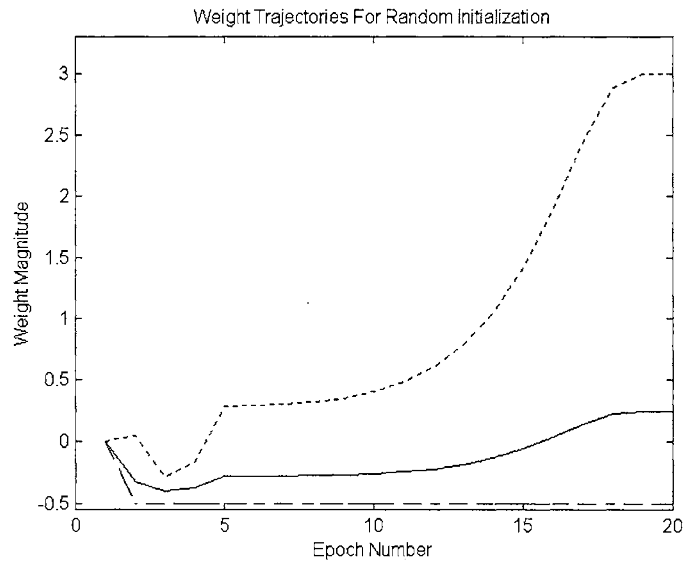


Figure 14 Weight Trajectories for Gauss-Newton Example

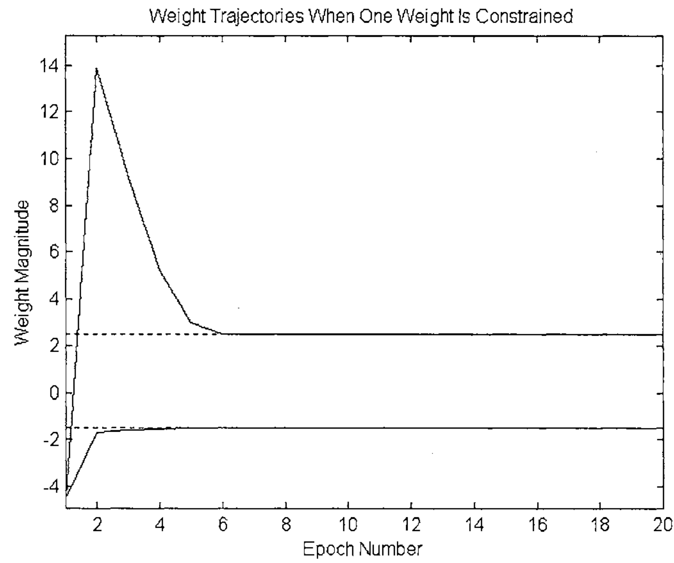


Figure 15 Weight Trajectories for 2-Weight Gauss-Newton Example

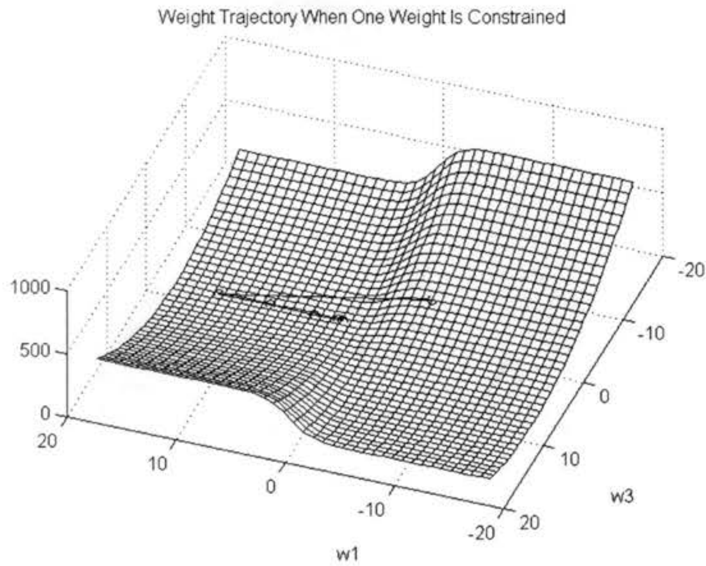


Figure 16 3-D Weight Trajectories for 3-Weight Gauss-Newton Example

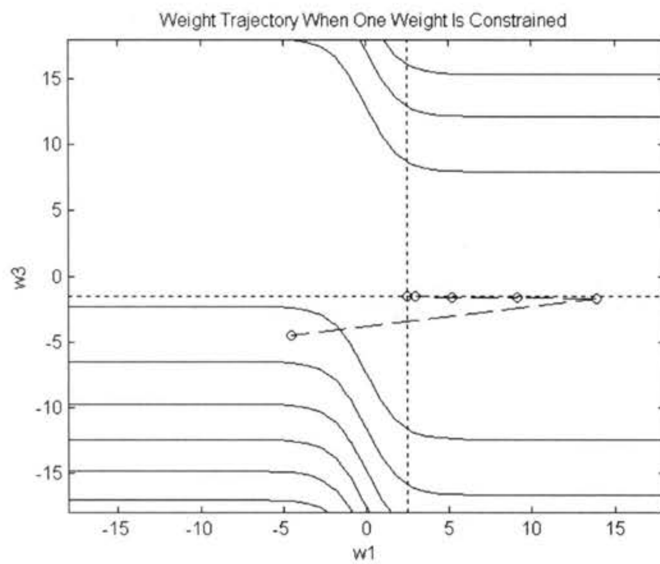


Figure 17 Weight Trajectories Contour Plot for 2-Weight Gauss-Newton Example

Levenberg-Marquardt Method

The Levenberg-Marquardt method [2] is a variation of the Gauss-Newton method in which the Hessian matrix is approximated by

$$\nabla^2 F(\mathbf{\Theta}) \approx 2\mathbf{J}^T(\mathbf{\Theta})\mathbf{J}(\mathbf{\Theta}) + 2\mu\mathbf{I} \tag{41}$$

Using Eq. (41) we can summarize the equations used in the Levenberg-Marquardt method as:

$$\Theta_{n+1} = \Theta_n + \Delta\Theta_{n_{OPT}} \tag{42}$$

$$\Delta\Theta_{n_{OPT}} = -[\nabla^2 F(\Theta)|_{\Theta = \Theta_n}]^{-1} \nabla F(\Theta)^T|_{\Theta = \Theta_n} \tag{43}$$

$$\nabla^2 F(\Theta) \approx 2\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + 2\mu\mathbf{I} \tag{44}$$

$$\nabla F(\Theta) = 2\mathbf{J}^T(\Theta)\mathbf{e}(\Theta) \tag{45}$$

or,

$$\Theta_{n+1} = \Theta_n - [\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + \mu\mathbf{I}]^{-1}\mathbf{J}^T(\Theta)\mathbf{e}(\Theta) \tag{46}$$

The $\mu \boldsymbol{I}$ term in Eq. (44) and Eq. (46) has some interesting effects on the parameter update computation. In the Gauss-Newton method, in which there is no $\mu\mathbf{I}$ term, $\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)$ may not be invertible. The $\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + \mu\mathbf{I}$ matrix will be invertible if μ is made large enough [2]. Additionally, as μ is increased, the weight update approaches the steepest descent method with small learning rate. This is a very nice quality in that the steepest descent method will always produce a decrease in the sum of the squared-errors if the learning rate is small enough. On the other hand, as μ is decreased, the Levenberg-Marquardt algorithm approaches the Gauss-Newton method, which under favorable conditions converges faster than steepest descent.

In the Levenberg-Marquardt algorithm, μ starts out set to a small number. If the sum of the squared-errors does not decrease, then μ is multiplied by a number greater than 1. This adjusts the algorithm towards steepest descent. This process is repeated until the

sum of the squared-errors decreases. If the sum of the squared-errors decreases, μ is divided by a number greater than 1. This adjusts the algorithm towards Gauss-Newton, which is generally faster than steepest descent. A mutilayer nonlinear network training example using the Levenberg-Marquardt method follows.

Levenberg-Marquardt Example

The Levenberg-Marquardt algorithm was applied to the network shown in Figure 7, which was trained in the previous examples. Again, 20 epochs of the algorithm were performed. Figure 18 shows a plot of the sum of the squared errors versus epoch number. The three unknown weights were initialized to small random numbers before training was performed.

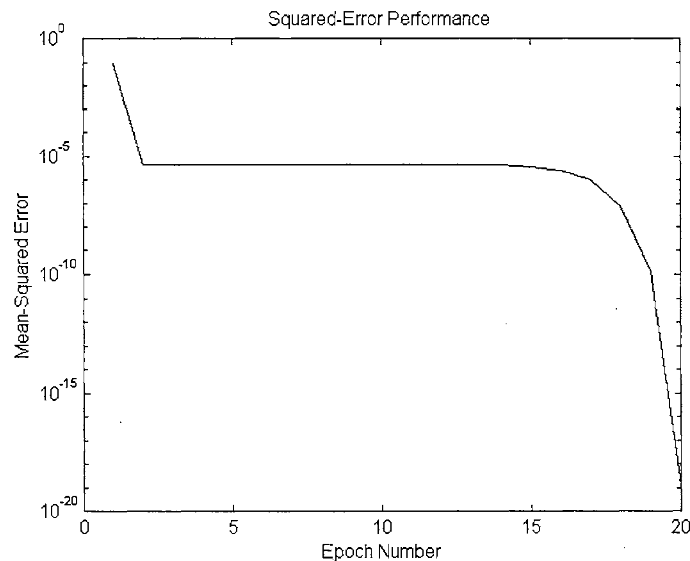


Figure 18 Squared-Error Performance for Levenberg-Marquardt Example

The trajectories of the unknown weights as training progresses are shown in Figure 19. In this plot is can be seen that after 20 epochs of training the weights have converged.

Notice how the Levenberg-Marquardt method converges with approximately the same speed as Newton's method did for this example, but much faster than steepest descent did. Next, all but two of the network weights were fixed, and the two weights adjusted using Levenberg-Marquardt. The network weights were set to 2.5, 3.0, and -1.5 to generate the training data. After initializing network weights to -4.5, 3.0 and -4.5, weights w_1 and w_3 were adjusted while w_2 was held fixed at it's original value. As expected, when the training was performed, the weights converged on the original values as shown in Figure 20. The trajectory for the weights are similar to those for Newton' method. Three-dimensional and contour plots of the weights traversing the squared-error surface are shown in Figure 21 and Figure 22. Notice in Figure 21 jumps to a new location as with the Newton and Gauss-Newton methods.

For the example problem shown here, Newton's method, Gauss-Newton and Levenberg-Marquardt all exhibit very similar performance. This is most likely because of the simplicity of the problem.

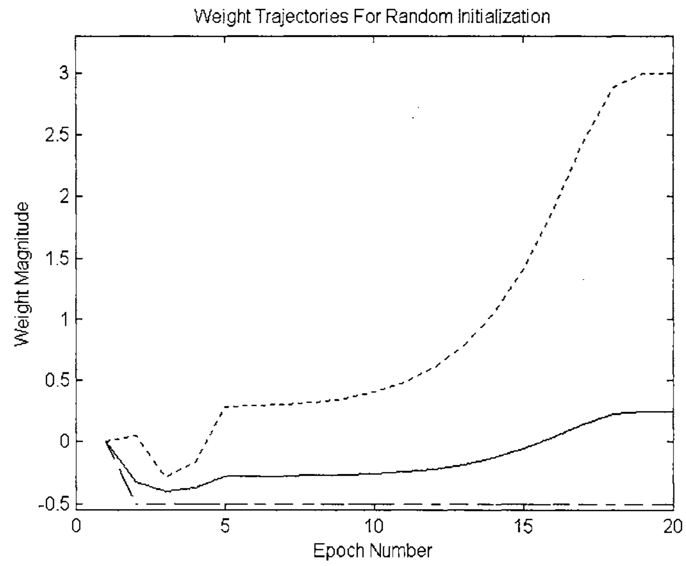


Figure 19 Weight Trajectories for Levenberg-Marquardt Example

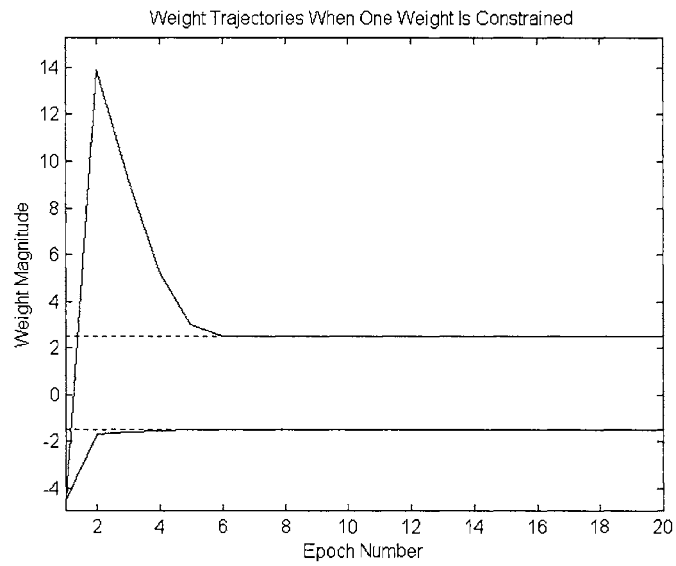


Figure 20 Weight Trajectories for 2-Weight Levenberg-Marquardt Example

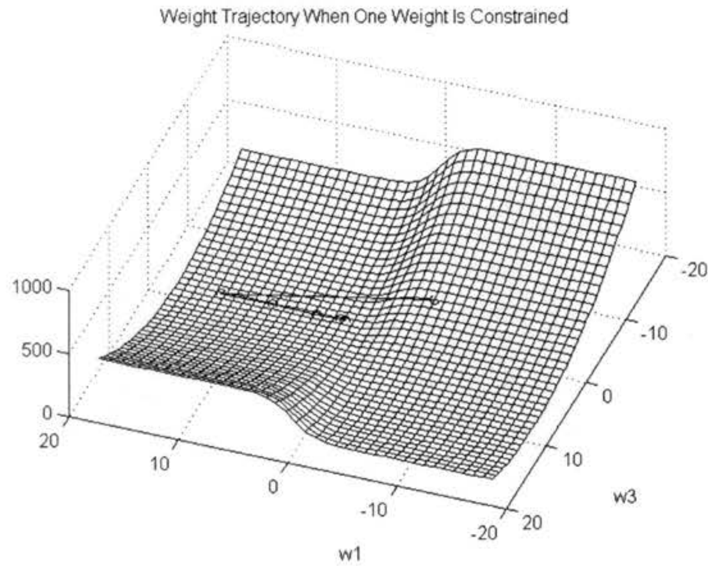


Figure 21 3-D Weight Trajectories for 3-Weight Levenberg-Marquardt Example

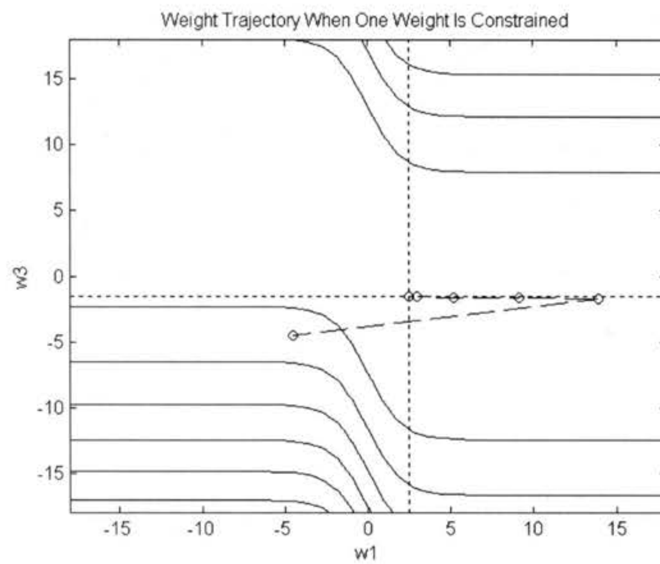


Figure 22 Weight Trajectories Contour Plot for 2-Weight Levenberg-Marquardt Example

Linear Least-Squares

One common method of estimating the unknown parameters of *linear* functions is the method of linear least squares [3]. In this method the function to be minimized is the familiar sum-of-squared-errors function given as:

$$F(\Theta) = \sum_{j=1}^{N} e_j^2(\hat{\Theta}) = \mathbf{e}^T(\hat{\Theta})\mathbf{e}(\hat{\Theta}) \tag{47}$$

where: $$\mathbf{e}(\hat{\Theta}) = [\mathbf{t} - \mathbf{a}(\hat{\Theta})] \tag{48}$$

In Eq. (47) the vector $\mathbf{e}(\hat{\Theta})$ represents the difference between a vector of desired outputs $\mathbf{t}$, and vector of outputs $\mathbf{a}(\hat{\Theta})$, produced by a linear function of the form,

$$a(n) = \hat{\Theta}_1 g_1(n) + \hat{\Theta}_2 g_2(n) + \ldots + \hat{\Theta}_M g_M(n) \tag{49}$$

where $\hat{\Theta}$ is a vector of parameter estimates, and $g_1(n), g_2(n), \ldots, g_M(n)$ are scalar inputs. These scalar inputs can be nonlinear functions of other systems inputs, as with radial basis function neural networks [4].

If we substitute Eq. (49) for all index values $(1 \ldots n \ldots N)$ into Eq. (48) and use Eq. (47), the following equation results:

$$F(\hat{\Theta}) = [\mathbf{t} - \mathbf{G}\hat{\Theta}]^T[\mathbf{t} - \mathbf{G}\hat{\Theta}] \tag{50}$$

where:

$$\mathbf{G} = \begin{bmatrix} g_1(1) & g_2(1) & \ldots & g_M(1) \\ g_1(2) & g_2(2) & \ldots & g_M(2) \\ \vdots & \vdots & \vdots & \vdots \\ g_1(N) & g_2(N) & \ldots & g_M(N) \end{bmatrix} \tag{51}$$

Eq. (50) can be rewritten as:

$$F(\hat{\Theta}) = \mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{G}\hat{\Theta} + \hat{\Theta}^T\mathbf{G}^T\mathbf{G}\hat{\Theta} \tag{52}$$

Taking the gradient of Eq. (52) with respect to Θ yields:

$$\nabla F(\hat{\Theta}) = -2[\mathbf{t}^T\mathbf{G}]^T + 2\mathbf{G}^T\mathbf{G}\hat{\Theta} \tag{53}$$

To find the parameters which minimize Eq. (52), we set the gradient equal to zero and solve for Θ. The resulting least squares estimate is

$$\hat{\Theta} = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{t} \tag{54}$$

Eq. (54) is only valid when the number of measurements available exceeds the number of parameters to be estimated. This situation is known as the *overdetermined* case [4]. When there are fewer measurements available than there are parameters to be estimated the situation can be described as the *underdetermined* case [4]. The solution to the underdetermined case is the central theme for a later chapter and will not be discussed further here.

Notice that, unlike the other minimization methods which have been discussed, the least-squares method is not iterative, but is solved in one step using a direct approach. This method is very useful in estimating unknown parameters in functions whose outputs are linearly related to the unknown parameters. This constraint renders this method of limited use in training neural networks, which often produce outputs which are not linearly related to the unknown network weights.

Nonlinear Least-Squares

The least-squares method may be used in minimizing nonlinear functions by first linearizing the function about a nominal set of parameter values, then applying the standard least-squares equations to calculate an incremental parameter adjustment. This process is repeated until a specified convergence criteria has been satisfied. This modification of the

least-squares method is known as the linearized least-squares method or the iterated least-squares method [3].

A development of the iterated least-squares method begins with the familiar equation,

$$F(\Theta) = \sum_{j=1}^{N} e_j^2(\hat{\Theta}) = \mathbf{e}^T(\hat{\Theta})\mathbf{e}(\hat{\Theta}) \tag{55}$$

where: $$\mathbf{e}(\hat{\Theta}) = [\mathbf{t} - \mathbf{a}(\hat{\Theta})] \tag{56}$$

which has been explained in the preceding sections. Now suppose that $\mathbf{a}(\Theta)$ is a vector of outputs of a nonlinear function of known inputs and unknown function parameters Θ, and $\mathbf{t}$ is a vector of desired outputs. Now we must linearize $\mathbf{a}(\Theta)$. The incremental change in the vector of function outputs for an incremental adjustment in the function parameters is represented by Eq. (57).

$$\mathbf{a}(\hat{\Theta}_{n+1}) = \mathbf{a}(\hat{\Theta}_n + \delta\hat{\Theta}_n) \tag{57}$$

In Eq. (57), $\delta\hat{\Theta}_n$ is the vector of incremental adjustments of the estimated parameters. Taking the first-order Taylor series expansion of Eq. (57) produces Eq. (58).

$$\mathbf{a}(\hat{\Theta}_{n+1}) \approx \mathbf{a}(\hat{\Theta}_n) + [\mathbf{J}(\hat{\Theta}_n)|_{\Theta = \Theta_n}]^T \delta\hat{\Theta}_n \tag{58}$$

where:

$$\mathbf{J}(\hat{\Theta}_n) = \begin{bmatrix} \frac{\partial}{\partial \hat{\Theta}_{1,n}} a_1(\hat{\Theta}_n) & \frac{\partial}{\partial \hat{\Theta}_{1,n}} a_2(\hat{\Theta}_n) & \cdots \frac{\partial}{\partial \hat{\Theta}_{1,n}} a_N(\hat{\Theta}_n) \\ \frac{\partial}{\partial \hat{\Theta}_{2,n}} a_1(\hat{\Theta}_n) & \frac{\partial}{\partial \hat{\Theta}_{2,n}} a_2(\hat{\Theta}_n) & \cdots \frac{\partial}{\partial \hat{\Theta}_{2,n}} a_N(\hat{\Theta}_n) \\ \vdots & \vdots & \vdots \\ \frac{\partial}{\partial \hat{\Theta}_{M,n}} a_1(\hat{\Theta}_n) & \frac{\partial}{\partial \hat{\Theta}_{M,n}} a_2(\hat{\Theta}_n) & \cdots \frac{\partial}{\partial \hat{\Theta}_{M,n}} a_N(\hat{\Theta}_n) \end{bmatrix}^T \tag{59}$$

In Eq. (59) M is the number of unknown parameters, N is the number of input/target-output data pairs, and n is the index which corresponds to the iteration of data set presentation.

Using Eq. (58), Eq. (55) can be written as:

$$F(\hat{\Theta}) \approx [\mathbf{t} - \mathbf{a}(\hat{\Theta}_n) + \mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n]^T[\mathbf{t} - \mathbf{a}(\hat{\Theta}_n) + \mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n] \tag{60}$$

Rewriting Eq. (60) in terms of the function output error we have:

$$F(\hat{\Theta}) \approx [\mathbf{e}(\hat{\Theta}_n) + \mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n]^T[\mathbf{e}(\hat{\Theta}_n) + \mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n] \tag{61}$$

Rearranging Eq. (61) we have:

$$F(\hat{\Theta}) \approx \mathbf{e}(\hat{\Theta}_n)^T\mathbf{e}(\hat{\Theta}_n) + 2\mathbf{e}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n + \delta\hat{\Theta}_n^T\mathbf{J}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)(\delta\hat{\Theta})_n \tag{62}$$

Taking the vector derivative of Eq. (62) with respect to $\delta\hat{\Theta}_n$ yields:

$$\nabla F(\hat{\Theta}) \approx 2[\mathbf{e}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)]^T + 2\mathbf{J}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)(\delta\hat{\Theta})_n \tag{63}$$

To find the changes in the parameters which minimize Eq. (62), we set the gradient equal to zero and solve for $\delta\hat{\Theta}_n$. The resulting linearized least-squares estimate of the incremental parameter adjustment is

$$\delta\hat{\Theta}_n = -[\mathbf{J}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)]^{-1}\mathbf{J}(\hat{\Theta}_n)^T\mathbf{e}(\hat{\Theta}_n) \tag{64}$$

In essence, the least-squares solution to the linearized system is computed about the current set of parameter estimates to find the appropriate vector of parameter adjustments, which are then used in Eq. (65) to compute an updated vector of parameter estimates.

$$\hat{\Theta}_{n+1} = \hat{\Theta}_n + \delta\hat{\Theta}_n \tag{65}$$

Eq. (64) can be inserted into Eq. (65) to obtain:

$$\hat{\Theta}_{n+1} = \hat{\Theta}_n - [\mathbf{J}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)]^{-1}\mathbf{J}(\hat{\Theta}_n)^T\mathbf{e}(\hat{\Theta}_n) \tag{66}$$

In a manner similar to the other methods presented in this chapter, Eq. (66) is applied repeatedly to a set of input/output data until some convergence criteria is satisfied. Note that Eq. (66) is equivalent to Eq. (40) for the Gauss-Newton Method.

Chapter 3

STANDARD INCREMENTAL OPTIMIZATION TECHNIQUES

Introduction

The purpose of this chapter is to present several numerical optimization techniques in which parameter updates are made each time a new data point becomes available rather than in a batch manner. This type of processing is desirable in "on-line" applications such as adaptive control, signal processing and system identification.

In Chapter 2 several methods of *batch* optimization were presented. In batch optimization all data is collected then processed simultaneously to compute a parameter update. The methods which will be presented here all have the characteristic of performing the computation of parameter updates each time a new sample is obtained. Some "on-line" methods use the parameter updates from the previous time-step to compute the updates at the current time-step making these methods *recursive* in nature.

All the "on-line" methods presented here use Eq. (1), rewritten below for convenience, to update the parameters Θ.

$$\Theta_{n+1} = \Theta_n + \Delta\Theta_{n_{OPT}} \tag{67}$$

The difference in the methods lies in how $\Delta\Theta_{n_{OPT}}$ is computed. A notable difference between "on-line" methods and batch methods lies in the number of times Eq. (67) is applied

to obtain a comparable level of optimization. In general, many more iterations of Eq. (67) are required for "on-line" optimization than for batch optimization.

The development of "on-line" or recursive forms of the methods of Steepest Descent, Linear Least Squares and Gauss-Newton will be given in this chapter. The formulation of the "on-line" methods will be presented as extensions of the development of the batch methods presented in Chapter 2. Additionally, a discussion of the Extended Kalman Filter (EKF) and its use in training neural networks will be presented. There are many optimization methods which are essentially modifications of the methods which will presented. Only the basic methods will be considered here. Illustrative examples in which the steepest descent, and recursive Gauss-Newton methods are applied to a simple neural network training problem will be given. A training example for the EKF algorithm will be omitted because of its equivalence to the recursive Gauss-Newton method. A training example using the linear recursive least-squares (RLS) algorithm will not be presented because the method is not suitable for training nonlinear neural networks.

Method of Steepest Descent (On-line Processing)

The most commonly used real-time optimization method is probably the steepest descent algorithm, in which an approximate gradient is used. This algorithm has been used successfully in many applications. The "real-time" version of the steepest descent algorithm can be written as:

$$\Theta_{k+1} = \Theta_k - \alpha[\hat{\nabla} F(\Theta)|_{\Theta = \Theta_k}] \tag{68}$$

The derivation of Eq. (68) was given in Chapter 2. Notice in Eq. (68) that an approximate gradient has replaced the true gradient in the batch steepest descent update algorithm given

by Eq. (6). This modification was first proposed by Widrow and Hoff [5]. They showed that the expected mean squared error for a series of data can be estimated as the squared error at a particular point in the data. This means that the gradient in a series of data can be estimated by the gradient at a particular point in the data. If an estimate of the gradient for each point in a series of data is made in this way, and the result for all points summed, the result will be equivalent to a batch computation of the gradient for the series of points. This type of gradient estimation over a series of data is known as a stochastic gradient computation. This makes it possible to use the gradient associated with the error at *each* point in a series of data to compute parameter updates. Therefore, in Eq. (68) the index k does not indicate an epoch number, but a sample number in a series of data. A neural network training example illustrating the use of the on-line steepest descent method will be presented in the next section.

Steepest Descent Example (On-line Processing)

For this example we will use the same network used for the batch training examples of Chapter 2. This network is shown in Figure 23. This example network has 1 input, 1 nonlinear layer, and a linear output layer which produces a single output. The nonlinear activation functions are log-sigmoid functions which are described by Eq. (11) and Eq. (12). There are three parameters, w_1, w_2 and w_3, which must be adjusted to appropriate values using the on-line steepest descent method.

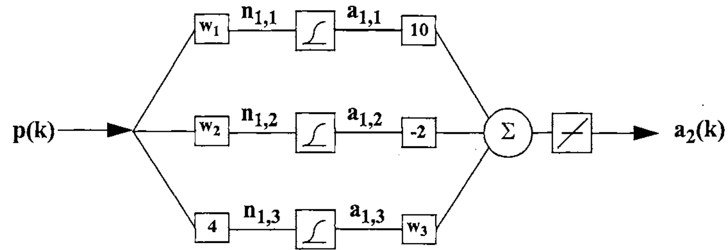


Figure 23 Example Nonlinear Neural Network

Test data were generated as in the previous examples by setting and fixing all the network weights to known values, then applying a set of inputs and recording the corresponding outputs. The on-line steepest descent method was then applied to the same network structure with the three unknown weights initialized to small random numbers. For this example a set of 1000 input/output pairs were used in training the network. This data set was processed using the on-line steepest descent algorithm.

Figure 24 shows a plot of the squared error as a function of sample number. The trajectories of the unknown weights are shown in Figure 25. We can see that the weights take a "noisy" path as the trajectories converge towards their final values. This is because the individual weight adjustments are based on estimates of the true gradient as described in the previous section. These small adjustments are in error on an individual basis, but as a whole converge to the proper values quite well.

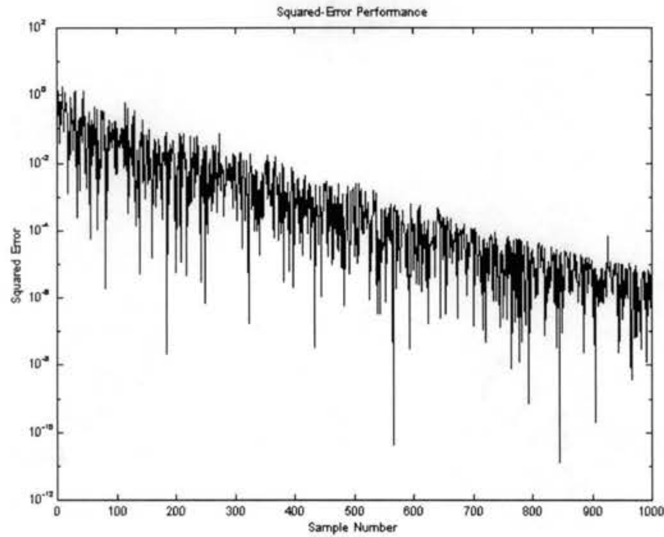


Figure 24 Squared-Error Performance for On-line Steepest Descent Example

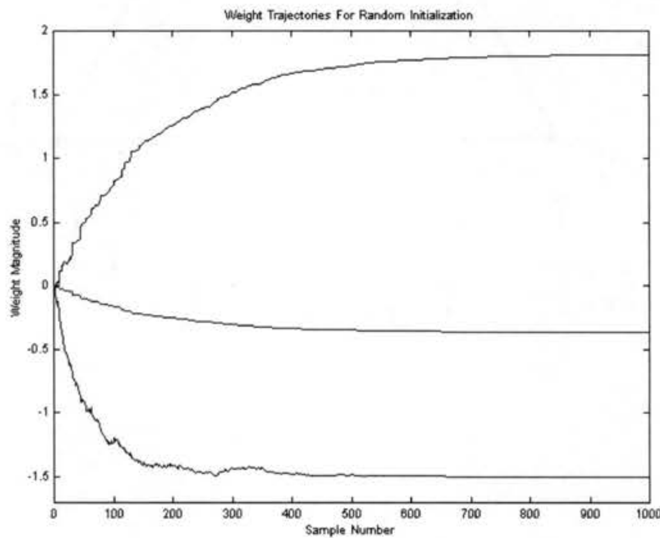


Figure 25 Weight Trajectories for On-line Steepest Descent Example

A second simulation was run for this example network in which all but two of the network weights were fixed, and then only these two weights adjusted. The network weights were set to 2.5, 3.0, and -1.5 to generate the training data. The network weights were initially set to -4.5, 3.0 and -4.5 and weights w_1 and w_3 adjusted while w_2 was held

fixed at its original value. By constraining the network in this way we should force the weights to return to the values used in generating the training data.

As expected, when the training was performed, the weights converged on the original values, as shown in Figure 26. Again, we see "noisy" weight trajectories because of the stochastic gradient used with this method. Three-dimensional and contour plots of the weights traversing the squared-error surface are shown in Figure 27 and Figure 28.

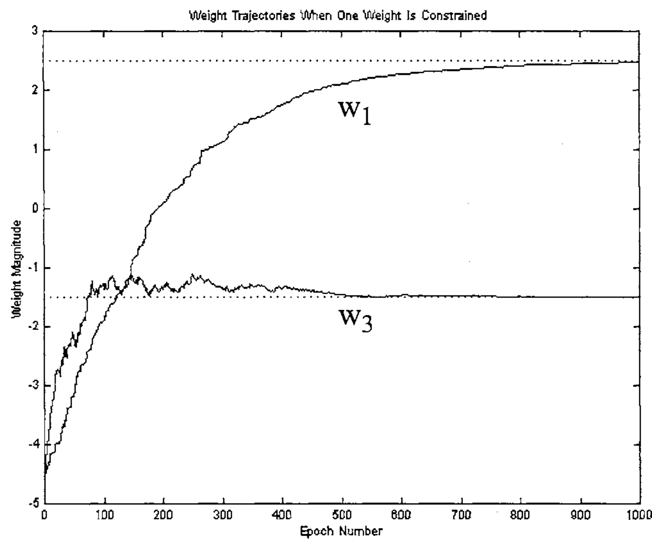


Figure 26 Weight Trajectories for 2-Weight On-line Steepest Descent Example

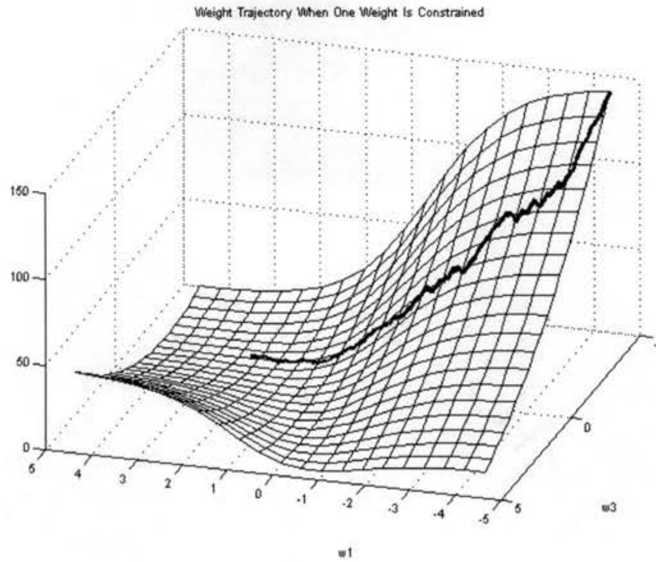


Figure 27 3-D Weight Trajectories for 2-Weight On-line Steepest Descent Example

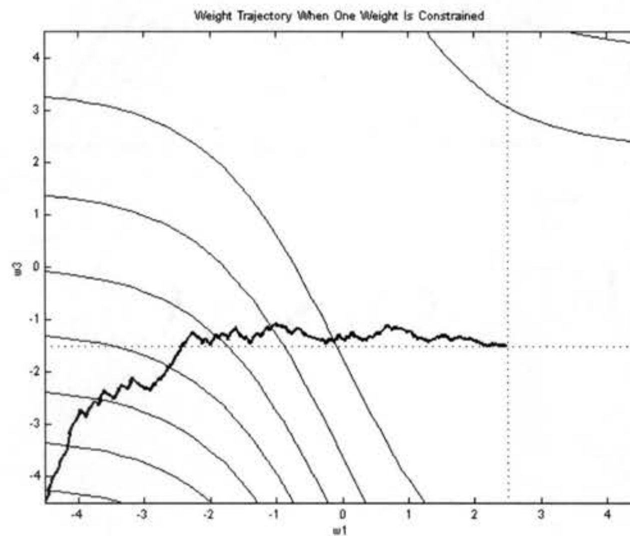


Figure 28 Weight Trajectories for 2-Weight On-line Steepest Descent Example

Figure 27 and Figure 28 show results similar to those obtained when this example was worked using the batch steepest descent method. The only noticeable difference is the slightly erratic weight trajectory path produced by the on-line algorithm.

Recursive Least-Squares

In this section a recursive method of computing the least squares solution to a linear function parameter estimation problem will be presented. Although this method is not suitable for training nonlinear neural networks, the development of this algorithm is the basis for many important recursive algorithms which are used in training nonlinear neural networks. For this reason the development of the recursive least-squares (RLS) algorithm will be presented in detail in this section. There will be no RLS example as it is only applicable to linear estimation.

The idea for the recursive algorithm is to use the parameter estimate $\hat{\Theta}$ produced by the previous *(n-1)* samples along with the data associated with the sample taken at time *(n)* to produce a new estimate of $\hat{\Theta}$. Having done this, we will have an algorithm which is suitable for on-line linear estimation. To begin the development of a recursive least squares algorithm for the single output case, we start with the batch linear least squares solution developed in Chapter 2 (Eq. (54)) which is:

$$\hat{\Theta} = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{t} \tag{69}$$

In Eq. (69) $\mathbf{G}$ is an N by M matrix of function inputs, where N is the number of input/output pairs used in estimating $\hat{\Theta}$, and M is the dimension of $\hat{\Theta}$. $\mathbf{G}$ is defined in Eq. (51).

For many applications it is common to incorporate a *weighting matrix* into the familiar sum-of-squared-errors function. The result is

$$F(\Theta) = \mathbf{e}^T(\hat{\Theta})\mathbf{W}\mathbf{e}(\hat{\Theta}) \tag{70}$$

where: $$\mathbf{e}(\hat{\Theta}) = [\mathbf{t} - \mathbf{a}(\hat{\Theta})] \tag{71}$$

The introduction of the weighting matrix $\mathbf{W}$ into Eq. (70) results in the well-known Weighted Least-Squares solution [3], which is

$$\hat{\Theta} = [\mathbf{G}^T\mathbf{W}\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{W}\mathbf{t} \tag{72}$$

$\mathbf{W}$ is symmetric and positive definite and is usually a diagonal matrix. A common choice for $\mathbf{W}$ is given in Eq. (73) for the single output case.

$$\mathbf{W} = \begin{bmatrix} \ddots & \vdots & \vdots & \vdots \\ \dots & \gamma^2 & 0 & 0 \\ \dots & 0 & \gamma & 0 \\ \dots & 0 & 0 & 1 \end{bmatrix} \tag{73}$$

If $|\gamma| < 1$, the most recent measurements are weighted more heavily than past ones. This choice of $\mathbf{W}$ effectively introduces a "forgetting" factor into the weighted least-squares solution. The importance of including a forgetting factor in the least-squares solution will become more apparent as the development of a recursive least-squares algorithm progresses.

Consider the situation where a current parameter estimate has been computed, and a new measurement has just become available. Starting now with Eq. (72) we can develop a recursive weighted least-squares algorithm. Eq. (72) can be rewritten as

$$\hat{\Theta}(k) = [\mathbf{G}(k)^T\mathbf{W}(k)\mathbf{G}(k)]^{-1}\mathbf{G}(k)^T\mathbf{W}(k)\mathbf{t}(k) \tag{74}$$

Now express $\hat{\Theta}(k)$ as

$$\hat{\Theta}(k) = \mathbf{P}(k)\mathbf{G}(k)^T\mathbf{W}(k)\mathbf{t}(k) \tag{75}$$

where

$$\mathbf{P}(k) = [\mathbf{G}(k)^T\mathbf{W}(k)\mathbf{G}(k)]^{-1} \tag{76}$$

Rearranging the terms in Eq. (75) results in

$$\mathbf{G}(k)^T\mathbf{W}(k)\mathbf{t}(k) = \mathbf{P}^{-1}(k)\hat{\Theta}(k) \tag{77}$$

Each new measurement adds an additional row $\mathbf{g}^T(k)$, to $\mathbf{G}(k)^T$. The number of elements in $\mathbf{g}^T(k)$ is equal to the number of function parameters. Appending $\mathbf{g}^T(k)$ to $\mathbf{G}(k)^T$ adds an outer product matrix to $[\mathbf{G}(k)^T\mathbf{W}(k)\mathbf{G}(k)]$. This allows Eq. (76) to be re-written as

$$\mathbf{P}^{-1}(k+1) = \mathbf{P}^{-1}(k) + \mathbf{g}(k+1)\gamma\mathbf{g}^T(k+1) \tag{78}$$

where γ is the scalar shown in Eq. (73).

Eq. (75) can be rewritten as

$$\hat{\Theta}(k+1) = \mathbf{P}(k+1)[\mathbf{g}(k+1)\gamma \mathrm{t}(k+1) + \mathbf{P}^{-1}(k)\hat{\Theta}(k)] \tag{79}$$

where $\mathrm{t}(k+1)$ is the function output target at index *k+1*. This equation can be rearranged as

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + \mathbf{K}(k+1)[\mathrm{t}(k+1) - \mathbf{g}^T(k+1)\hat{\Theta}(k)] \tag{80}$$

where

$$\mathbf{K}(k+1) = \mathbf{P}(k+1)\mathbf{g}(k+1)\lambda \tag{81}$$

and

$$\mathbf{P}^{-1}(k+1) = \mathbf{P}^{-1}(k) + \mathbf{g}(k+1)\gamma\mathbf{g}^T(k+1) \tag{82}$$

Eq. (80), Eq. (81) and Eq. (82) represent a recursive formulation of the weighted least squares solution. The implementation of these equations proceeds as follows:

$$\mathbf{P}^{-1}(k+1) \rightarrow \mathbf{P}(k+1) \rightarrow \mathbf{K}(k+1) \rightarrow \hat{\Theta}(k+1)$$

In order to implement this recursive formulation, it is necessary to invert $\mathbf{P}^{-1}(k+1)$. Performing this inversion is computationally intensive, and is therefore undesirable. Fortunately, we can use the well-known *Matrix Inversion Lemma* [3] to avoid having to perform this inversion. A statement of the matrix inversion lemma follows.

Matrix Inversion Lemma

If the matrices **A**, **B**, **C**, *and* **D** *satisfy the equation*

$$\mathbf{B}^{-1} = \mathbf{A}^{-1} + \mathbf{C}^{T}\mathbf{D}^{-1}\mathbf{C} \tag{83}$$

where all matrix inverses are assumed to exist, then

$$\mathbf{B} = \mathbf{A} - \mathbf{A}\mathbf{C}^{T}(\mathbf{C}\mathbf{A}\mathbf{C}^{T} + \mathbf{D})^{-1} \tag{84}$$

A detailed proof of the matrix inversion lemma may be found in [3].

To avoid inverting $\mathbf{P}^{-1}(k+1)$, we can apply the matrix inversion lemma to Eq. (82) to obtain

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{P}(k)\mathbf{g}(k+1)[\mathbf{g}^{T}(k+1)\mathbf{P}(k)\mathbf{g}(k+1) + (1/\gamma)]^{-1}\mathbf{g}^{T}(k+1)\mathbf{P}(k) \tag{85}$$

This still requires the inversion of $[\mathbf{g}^{T}(k+1)\mathbf{P}(k)\mathbf{g}(k+1) + (1/\gamma)]$, but in the single output case this is a simple scalar. Even in the case of multiple outputs the size of the matrix which must be inverted is equal only to the number of function outputs. Generally, the number of function outputs will be much smaller than the number of parameters. Remem-

ber that the implementation of the RLS algorithm without the use of the matrix inversion lemma requires the inversion of a matrix which has a size proportional to the number of parameters. The combination of Eq. (80), Eq. (81) and Eq. (85) comprise an efficient implementation of the RLS algorithm which may be written as

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + \mathbf{K}(k+1)[\mathrm{t}(k+1) - \mathbf{g}^T(k+1)\hat{\Theta}(k)] \tag{86}$$

where

$$\mathbf{K}(k+1) = \mathbf{P}(k+1)\mathbf{g}(k+1)\gamma \tag{87}$$

and

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{P}(k)\mathbf{g}(k+1)[\mathbf{g}^T(k+1)\mathbf{P}(k)\mathbf{g}(k+1) + (1/\gamma)]^{-1}\mathbf{g}^T(k+1)\mathbf{P}(k) \tag{88}$$

The recursive RLS algorithm can be used for parameter estimation in multiple output functions. The development is similar to that for the single output case. The RLS equations for the multiple output case are

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + \mathbf{K}(k+1)[\mathbf{t}_k(k+1) - \mathbf{g}^T(k+1)\hat{\Theta}(k)] \tag{89}$$

where

$$\mathbf{K}(k+1) = \mathbf{P}(k+1)\mathbf{g}(k+1)\gamma(k+1) \tag{90}$$

and

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{P}(k)\mathbf{g}(k+1)[\mathbf{g}^T(k+1)\mathbf{P}(k)\mathbf{g}(k+1) + \gamma(k+1)^{-1}]^{-1}\mathbf{g}^T(k+1)\mathbf{P}(k) \tag{91}$$

For the multiple output case $\mathbf{g}^T(k+1)$ is a rectangular matrix. The number of rows in $\mathbf{g}^T(k+1)$ is equal to the number of function outputs, and the number of columns is equal to the number of function parameters. $\gamma(k+1)$ is a diagonal matrix with the forgetting factor γ on the diagonal. The size of $\gamma(k+1)$ is equal to the number of function outputs. $\mathbf{t}(k+1)$ is the vector of function output targets at index $k+1$.

The implementation of Eq. (86), Eq. (87) and Eq. (88) or Eq. (89), Eq. (90) and Eq. (91) proceeds as follows:

$$\mathbf{P}(k) \rightarrow \mathbf{K}(k+1) \rightarrow \hat{\Theta}(k+1) \rightarrow \mathbf{P}(k+1)$$

$\mathbf{P}(k)$ is usually initialized to a diagonal matrix of large numbers, and is often periodically reset during operation to ensure numerical stability. The RLS algorithm is only suitable for use with functions which have outputs which are linearly related to the parameters. In the next section we will see how the development presented here for the RLS algorithm leads directly to recursive form of the Gauss-Newton method which is applicable to nonlinear parameter estimation.

Recursive Gauss-Newton Method

In Chapter 2 the batch Gauss-Newton method was presented. In this section a recursive form of the Gauss-Newton method (RGN) will be presented. It was shown that the linearized least squares method is equivalent to the Gauss-Newton method and therefore the separate development of a recursive linearized least-squares algorithm is not necessary. As with the RLS algorithm, it will be shown than an efficient form of the RGN method which utilizes the matrix inversion lemma can be developed.

Recalling from Chapter 2, the parameter update equation for the batch Gauss-Newton method is

$$\Theta_{n+1} = \Theta_n - [\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)]^{-1}\mathbf{J}^T(\Theta)\mathbf{e}(\Theta) \tag{92}$$

where $\mathbf{J}$ is the Jacobian matrix and $\mathbf{e}(\Theta)$ is the function output error. The index n in Eq. (92) refers to the data epoch number. Now suppose we rewrite Eq. (92) in a form where it is assumed that the Jacobian matrix and error vector are functions of the parameters, and where all elements of the equation are indexed in terms of sample number instead of epoch number. The result is

$$\Theta(k+1) = \Theta(k) - [\mathbf{J}^T(k)\mathbf{J}(k)]^{-1}\mathbf{J}^T(k)\mathbf{e}(k) \tag{93}$$

The parameter update in Eq. (93) is

$$\Delta\Theta(k) = -[\mathbf{J}^T(k)\mathbf{J}(k)]^{-1}\mathbf{J}^T(k)\mathbf{e}(k) \tag{94}$$

If we add a weighting matrix as described previously in the section on the RLS algorithm to Eq. (94) we have

$$\Delta\Theta(k) = -[\mathbf{J}^T(k)\mathbf{W}(k)\mathbf{J}(k)]^{-1}\mathbf{J}^T(k)\mathbf{W}(k)\mathbf{e}(k) \tag{95}$$

If the weighting matrix in Eq. (95) is of the form (for the single output case)

$$\mathbf{W}(k) = \begin{bmatrix} \ddots & \vdots & \vdots & \vdots \\ \cdots & \gamma^2 & 0 & 0 \\ \cdots & 0 & \gamma & 0 \\ \cdots & 0 & 0 & 1 \end{bmatrix} \tag{96}$$

past values will eventually be "forgotten", and recent values heavily weighted.

A close examination of Eq. (95) reveals that its form is exactly the same as Eq. (74) which is the batch solution to the linear weighted least squares problem. This means that the development of a recursive method of computing Eq. (94) can proceed in exactly the same way as was shown in the last section for the RLS solution. The details of this development will not be presented here but can easily be deduced by reviewing the previous section. The recursive Gauss-Newton (RGN) method may be summarized as

$$\Theta(k+1) = \Theta(k) - \Delta\Theta(k+1) \tag{97}$$

$$\Delta\Theta(k+1) = \Delta\Theta(k) + \mathbf{K}(k+1)[\mathbf{e}(k+1) - \mathbf{j}^T(k+1)\Delta\Theta(k)]$$

where

$$\mathbf{K}(k+1) = \mathbf{P}(k+1)\mathbf{j}(k+1)\gamma(k+1) \tag{98}$$

and

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{P}(k)\mathbf{j}(k+1)[\mathbf{j}^T(k+1)\mathbf{P}(k)\mathbf{j}(k+1) + \gamma(k+1)^{-1}]^{-1}\mathbf{j}^T(k+1)\mathbf{P}(k) \tag{99}$$

In Eq. (97), Eq. (98), and Eq. (99) $\mathbf{j}^T(k+1)$ is the matrix containing the newest rows of the Jacobian matrix, $\gamma(k+1)$ is a diagonal matrix with the forgetting factor γ on the diagonal and $\mathbf{e}(k)$ is a vector of function output errors. The number of rows contained in $\mathbf{j}^T(k+1)$ is equal to the number of function outputs. The size of $\gamma(k+1)$ is equal to the number of function outputs. The implementation of Eq. (86), Eq. (87) and Eq. (88) proceeds as follows:

$$\mathbf{P}(k) \rightarrow \mathbf{K}(k+1) \rightarrow \Delta\hat{\Theta}(k+1) \rightarrow \Theta(k+1) \rightarrow \mathbf{P}(k+1)$$

As in the standard RLS algorithm, $\mathbf{P}(k)$ is usually initialized to a diagonal matrix of large numbers, and is often periodically reset during operation to ensure numerical stability. The use of the RGN algorithm will now be illustrated by way of a simple neural network training example.

Recursive Gauss-Newton Method Example

In this section the RGN method will be demonstrated by using the algorithm to train the example network shown in Figure 23. Training data was generated by fixing the network weights and applying random inputs. The network weights were then initialized to small random values and the recorded training data presented during an adaptive training session.

Figure 29 shows a plot of the sum of the squared errors for 1000 data points. Notice how much more quickly the error is reduced for the RGN method when compared to the Steepest descent method (Figure 24). It is also notable how much lower the squared error is after 1000 points than was achieved with the steepest decent method. Figure 30 shows the trajectories of the weights during the training session. As expected, the weights converge to final values much more quickly than was demonstrated with the steepest descent method (Figure 25). The "noisy" trajectories result from the stochastic approximation of the gradient used at each iteration.

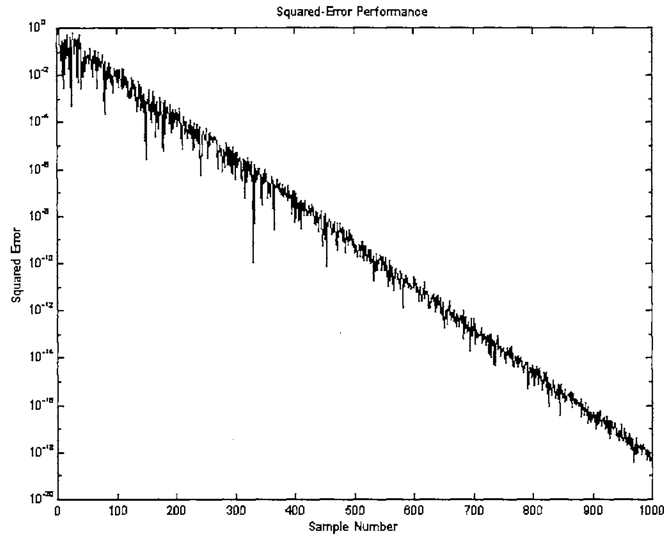


Figure 29 Squared-Error Performance for Recursive Gauss-Newton Example

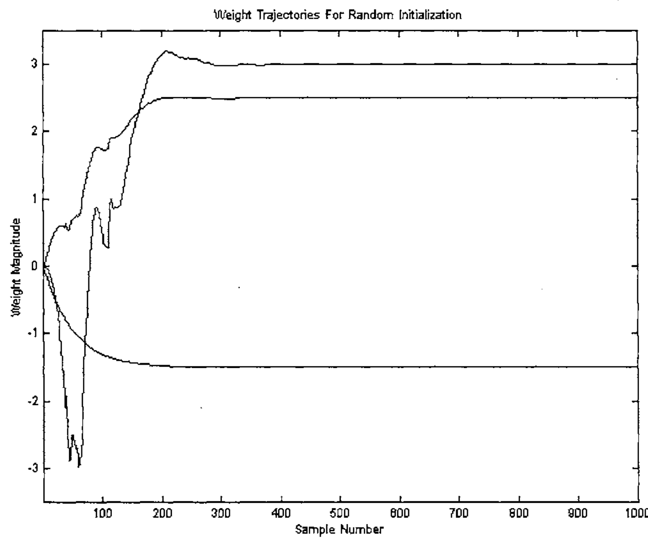


Figure 30 Weight Trajectories for Recursive Gauss-Newton Example

To improve the graphical illustration of the RGN method, all but two of the network weights were fixed, and these two weights were adjusted using the method. Fixed network weights of 2.5, 3.0, and -1.5 were used to generate training data. As in previous examples, the network weights were then initialized to -4.5, 3.0 and -4.5, and weights w_1 and w_3 were

adjusted using the RGN method, while w_2 was left at its original value. The resulting weight trajectories can be seen in Figure 31. The results show fast, stable convergence for this problem. Three-dimensional and contour plots of the weights traversing the squared-error surface are shown in Figure 32 and Figure 33. Notice in Figure 33 that the trajectory goes directly to the minimum point without following the gradient.

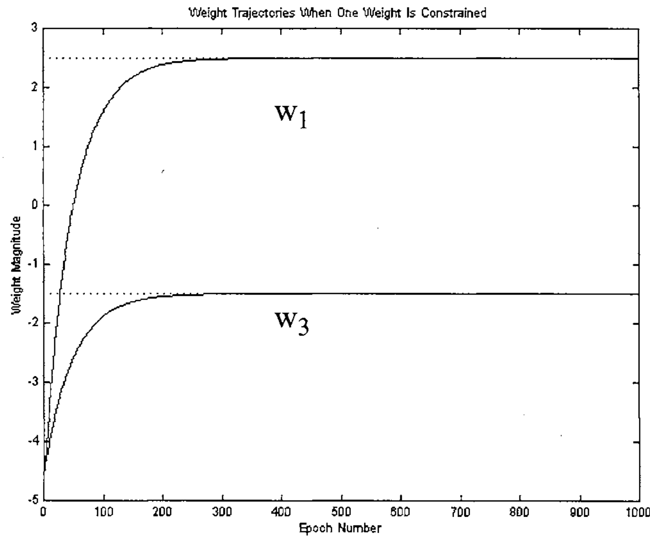


Figure 31 Weight Trajectories for 2-Weight Recursive Gauss-Newton Example

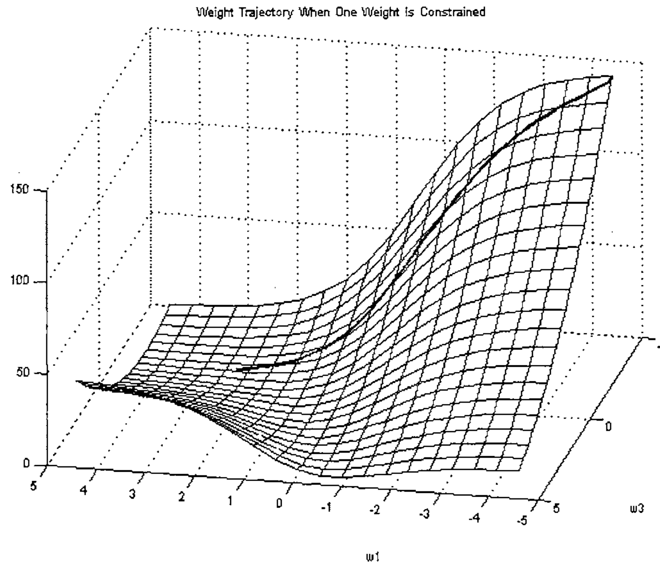


Figure 32 3-D Weight Trajectories for 3-Weight Recursive Gauss-Newton Example

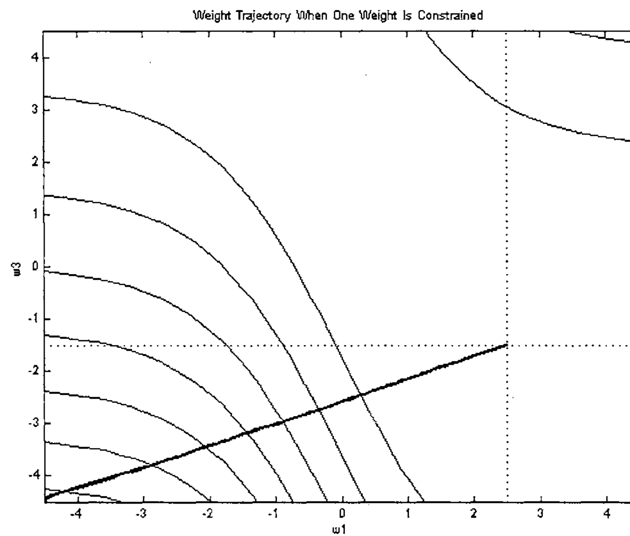


Figure 33 Weight Trajectories Contour Plot for 2-Weight RGN Example

Extended Kalman Filter

A very important algorithm which is widely used in control theory and signal processing is the Kalman Filter [6]. The standard Kalman filter is described mathematically

in terms of *state-space* concepts applied to *linear, discrete-time dynamical* systems. Sayed and Kailath [7] have shown that the Kalman filter provides a general framework for deriving all of the known algorithms that comprise the least-squares family of adaptive filters.

The Extended Kalman Filter (EKF) algorithm was first applied to parameter estimation in linear, state-space systems containing missing parameters in the state transition matrix and measurement matrix [8]. This work naturally led to the application of the EKF for state and parameter estimation in nonlinear models including nonlinear neural networks. Singhal and Wu [9] showed that the EKF algorithm could be used to train multi-layer-perceptron networks by treating the weights of the network as states in an unforced nonlinear dynamical system. This approach leads to a training algorithm which is identical to the recursive linearized least-squares algorithm which can be developed by applying the matrix inversion lemma to the batch linearized least-squares update equation (Eq. (64), Chapter 2). This result is stated by Pukorius and Feldcamp in [12]. Later researchers Matthews [10] and Williams [11] used the EKF algorithm to estimate both the states (node outputs) and the weights in a neural network. Williams [11] work was directed at training recurrent networks. This work was closely followed by Puskorius and Felkamp [12], [13], [14], [15] who have performed extensive research on using the EKF to train recurrent networks in a variety of adaptive control applications.

An important feature of the Kalman filter is that its solution is computed *recursively* using the previous solution along with the most recent input data. This means that it is necessary only to store the last solution in a computer implementation of the algorithm. As with the RLS algorithm, the Kalman filter solution can be arrived at with less computation

than computing a solution directly from past observed data. In this section a brief presentation of the Kalman filter for linear systems will be given and will be followed by a discussion of the Extended Kalman Filtering (EKF) algorithm for use with nonlinear systems. Finally, the application of the EKF algorithm to neural network training will be explained.

To begin our discussion of the Kalman filter, consider a linear, discrete-time dynamical system described by the following equations:

1. *Process Equation*

$$\mathbf{x}(k+1) = \mathbf{\Phi}(k)\mathbf{x}(k) + \mathbf{v}_1(k) \tag{100}$$

In Eq. (100) $\mathbf{x}(k)$ is a vector of system states and $\mathbf{\Phi}(k)$ is a M-by-M state transition matrix which relates the states of the system at time k to the states of the system at time $k+1$. $\mathbf{v}_1(k)$ is a M-by-1 vector which represents process noise. The vector $\mathbf{v}_1(k)$ is generally considered to be a zero-mean, white noise process which has a diagonal correlation matrix $\mathbf{Q}_1(k)$.

2. *Measurement Equation*

$$\mathbf{y}(n) = \mathbf{C}(k)\mathbf{x}(k) + \mathbf{v}_2(k) \tag{101}$$

In Eq. (101) $\mathbf{C}(k)$ is a N-by-M measurement matrix which relates the states, $\mathbf{x}(k)$ to the outputs $\mathbf{y}(k)$. $\mathbf{v}_2(k)$ is a N-by-1 vector which represents measurement noise. The vector $\mathbf{v}_2(k)$ is generally considered to be a zero-mean, white noise process which has a diagonal correlation matrix $\mathbf{Q}_2(k)$.

The purpose of the Kalman filtering algorithm is to use all the observed data $\mathbf{y}(1)$, $\mathbf{y}(2)$, $\mathbf{y}(3)$, . . ., $\mathbf{y}(k)$ to compute minimum-mean-square estimates of the states $\mathbf{x}(m)$.

For the distinct cases of $m > k$, $m = k$ and $m < k$ the problem can be classified as *prediction, filtering* or *smoothing,* respectively. In the work presented here, we will only be interested in the prediction problem.

The Kalman filter is well-known and its derivation has been widely published. This development will not repeated here, but is presented in detail by Haykin in [4]. The algorithm can be summarized as

$$\mathbf{K}(k) = \mathbf{\Phi}(k)\mathbf{P}(k)\mathbf{C}^T(n)[\mathbf{C}(k)\mathbf{P}(k)\mathbf{C}^T(k) + \mathbf{Q}_2(k)]^{-1} \tag{102}$$

$$\hat{\mathbf{x}}(k+1|y_k) = \mathbf{\Phi}(k)\hat{\mathbf{x}}(k|y_{k-1}) + \mathbf{K}(k)[\mathbf{y}(k) - \mathbf{C}(k)\hat{\mathbf{x}}(k|y_{k-1})] \tag{103}$$

$$\mathbf{P}(k+1) = \mathbf{\Phi}(k)[\mathbf{P}(k) - \mathbf{\Phi}(k)\mathbf{K}(k)\mathbf{C}(k)\mathbf{P}(k)]\mathbf{\Phi}^T(k) + \mathbf{Q}_1(k) \tag{104}$$

where:

$\mathbf{x}(k)$	State vector at time k
$\mathbf{y}(k)$	Observation vector at time k
$\mathbf{\Phi}(k)$	State transition matrix from time k to time $k+1$
$\mathbf{C}(k)$	Measurement matrix at time k
$\mathbf{Q}_1(k)$	Correlation matrix of process noise vector $\mathbf{v}_1(k)$
$\mathbf{Q}_2(k)$	Correlation matrix of measurement noise vector $\mathbf{v}_2(k)$
$\hat{\mathbf{x}}(k\|y_{k-1})$	Estimate of the state vector at time k given all past observations
$\mathbf{K}(k)$	Kalman gain at time k
$\mathbf{P}(k+1)$	Correlation matrix of the error in $\hat{\mathbf{x}}(k+1)$

The implementation of the algorithm proceeds as:

$$\mathbf{P}(k) \rightarrow \mathbf{K}(k) \rightarrow \hat{\mathbf{x}}(k+1|y_k) \rightarrow \mathbf{P}(k+1)$$

The use of standard Kalman filtering algorithm for linear systems can be "extended" for use in state and parameter estimation problems involving functions in which the states or parameters being estimated are related nonlinearly. This modified Kalman filtering algorithm is known as the Extended Kalman Filter (EKF) [3]. It is suitable for nonlinear systems of the form

$$\mathbf{x}(k+1) = \mathbf{\Phi}(k, \mathbf{x}(k)) + \mathbf{v}_1(k) \tag{105}$$

and

$$\mathbf{y}(k) = \mathbf{C}(k, \mathbf{x}(k)) + \mathbf{v}_2(k) \tag{106}$$

In Eq. (105), $\mathbf{x}(k)$ is a vector of system states and $\mathbf{\Phi}(k, \mathbf{x}(k))$ is a nonlinear transition function which relates the states of the system at time k to the states of the system at time $k+1$. As in the linear case, $\mathbf{v}_1(k)$ is a M-by-1 process noise vector of zero-mean, white noise which has a diagonal correlation matrix $\mathbf{Q}_1(k)$. In Eq. (106), $\mathbf{C}(k, \mathbf{x}(k))$ is a nonlinear measurement function which relates the states, $\mathbf{x}(k)$ to the outputs $\mathbf{y}(k)$. Again, $\mathbf{v}_2(k)$ is a N-by-1 vector of measurement noise which is generally considered to be a zero-mean, white noise and has a diagonal correlation matrix $\mathbf{Q}_2(k)$.

The key idea behind the EKF algorithm is to *linearize* the state-space system described by Eq. (105) and Eq. (106) at each time instant around the current state estimate $\mathbf{x}(k)$. Once the system is linearized, the standard Kalman filtering equations can be applied.

The first step in the linearization process is to construct the linearized matrices given by

$$\Phi(k) = \left.\frac{\partial\Phi(k, \mathbf{x}(k))}{\partial\mathbf{x}(k)}\right|_{\mathbf{x}(k) = \hat{\mathbf{x}}(k|y_k)} \tag{107}$$

and

$$\mathbf{C}(k) = \left.\frac{\partial\mathbf{C}(k, \mathbf{x}(k))}{\partial\mathbf{x}(k)}\right|_{\mathbf{x}(k) = \hat{\mathbf{x}}(k|y_{k-1})} \tag{108}$$

The next step is to use the result of Eq. (107) and Eq. (108) in a first-order Taylor approximation of $\Phi(k, \mathbf{x}(k))$, and $\mathbf{C}(k, \mathbf{x}(k))$ around $\hat{\mathbf{x}}(k|y_k)$, and $\hat{\mathbf{x}}(k|y_{k-1})$. The resulting approximations for $\Phi(k, \mathbf{x}(k))$, and $\mathbf{C}(k, \mathbf{x}(k))$ are

$$\Phi(k, \mathbf{x}(k)) \cong \Phi(k, \hat{\mathbf{x}}(k|y_k)) + \Phi(k)[\mathbf{x}(k) - \hat{\mathbf{x}}(k|y_k)] \tag{109}$$

and

$$\mathbf{C}(k, \mathbf{x}(k)) \cong \mathbf{C}(k, \hat{\mathbf{x}}(k|y_{k-1})) + \mathbf{C}(k)[\mathbf{x}(k) - \hat{\mathbf{x}}(k|y_{k-1})] \tag{110}$$

Substituting Eq. (109), and Eq. (110) into Eq. (105), and Eq. (106) respectively, results in a linearized system approximation which is

$$\mathbf{x}(k+1) \cong \Phi(k)\mathbf{x}(k) + \mathbf{v}_1(k) + [\Phi(k, \hat{\mathbf{x}}(k|y_k)) - \Phi(k)\hat{\mathbf{x}}(k|y_k)] \tag{111}$$

and

$$\mathbf{y}(k) \cong \mathbf{C}(k)\mathbf{x}(k) + \mathbf{v}_2(k) + [\mathbf{C}(k, \hat{\mathbf{x}}(k|y_{k-1})) - \mathbf{C}(k)\hat{\mathbf{x}}(k|y_{k-1})] \tag{112}$$

The system described by Eq. (111) and Eq. (112) is a linear system to which the standard Kalman filtering equations (Eq. (102) - Eq. (104)) may applied. In doing this and

applying some matrix algebra we arrive at the extended Kalman filtering algorithm which may be summarized as

$$\mathbf{K}(k) = \mathbf{P}(k)\mathbf{C}^T(k)[\mathbf{C}(k)\mathbf{P}(k)\mathbf{C}^T(k) + \mathbf{Q}_2(k)]^{-1} \tag{113}$$

$$\hat{\mathbf{x}}(k|y_k) = \hat{\mathbf{x}}(k|y_{k-1}) + \mathbf{K}(k)[\mathbf{y}(k) - \mathbf{C}(k, \hat{\mathbf{x}}(k|y_{k-1}))] \tag{114}$$

$$\hat{\mathbf{x}}(k+1|y_k) = \mathbf{\Phi}(k, \hat{\mathbf{x}}(k|y_k)) \tag{115}$$

$$\mathbf{P}(k+1) = \mathbf{\Phi}(k)[\mathbf{P}(k) - \mathbf{K}(k)\mathbf{C}(k)\mathbf{P}(k)]\mathbf{\Phi}^T(k) + \mathbf{Q}_1(k) \tag{116}$$

where:

$\mathbf{x}(k)$	State vector at time k
$\mathbf{y}(k)$	Observation vector at time k
$\mathbf{\Phi}(k, \mathbf{x}(k))$	Nonlinear state transition function
$\mathbf{C}(k, \mathbf{x}(k))$	Nonlinear measurement function
$\mathbf{\Phi}(k)$	Linearized state transition matrix from time k to time $k+1$
$\mathbf{C}(k)$	Linearized measurement matrix at time k
$\mathbf{Q}_1(k)$	Correlation matrix of process noise vector $\mathbf{v}_1(k)$
$\mathbf{Q}_2(k)$	Correlation matrix of measurement noise vector $\mathbf{v}_2(k)$
$\hat{\mathbf{x}}(k\|y_{k-1})$	Estimate of the state vector at time k given all *past* observations
$\hat{\mathbf{x}}(k\|y_k)$	Estimate of the state vector at time k given all observations
$\mathbf{K}(k)$	Kalman gain at time k
$\mathbf{P}(k+1)$	Correlation matrix of the error in $\hat{\mathbf{x}}(k+1)$

The implementation of the algorithm proceeds as:

$$\mathbf{P}(k) \rightarrow \mathbf{K}(k) \rightarrow \hat{\mathbf{x}}(k+1|y_k) \rightarrow \hat{\mathbf{x}}(k|y_k) \rightarrow \mathbf{P}(k+1)$$

The development of an EKF-based neural network training algorithm can now proceed by applying Eq. (113) - Eq. (116) to the nonlinear function $\mathbf{F}(\Theta(k), \mathbf{p}(k))$ which represents a nonlinear neural network. We must first relate the components of the network to the specific parts of the EKF algorithm. The weights $\Theta(k)$, of the network are considered "states" $\mathbf{x}(k)$, in the EKF algorithm, and the network function $\mathbf{F}(\Theta(k), \mathbf{p}(k))$, which relates the states to the output is considered the nonlinear measurement function $\mathbf{C}(k, \mathbf{x}(k))$. Because we know that the weights in a trained system model are constants, the state transition function $\Phi(k, \mathbf{x}(k))$ is a simple identity matrix.

Given these observations we can write the nonlinear state model for the neural network in terms of the EKF algorithm as

$$\Theta(k+1) = \Theta(k) + \mathbf{v}_1(k) \quad (117)$$

and

$$\mathbf{t}(k) = \mathbf{F}(\Theta(k), \mathbf{p}(k)) + \mathbf{v}_2(k) \quad (118)$$

In Eq. (118), $\mathbf{F}(\Theta(k), \mathbf{p}(k))$ is a nonlinear function of the network parameters $\Theta(k)$, and inputs $\mathbf{p}(k)$. This function is the neural network model of the nonlinear system. The application of the EKF algorithm to the system described by Eq. (117) and Eq. (118) yields

$$\mathbf{K}(k) = \mathbf{P}(k)\frac{\partial \mathbf{F}}{\partial \Theta_k}^T\left[\frac{\partial \mathbf{F}}{\partial \Theta_k}\mathbf{P}(k)\frac{\partial \mathbf{F}}{\partial \Theta_k}^T + \mathbf{Q}_2(k)\right]^{-1} \quad (119)$$

$$\Theta(k+1) = \Theta(k) + \mathbf{K}(k)[\mathbf{t}(k) - \mathbf{F}(\Theta(k), \mathbf{p}(k))] \quad (120)$$

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{K}(k)\frac{\partial \mathbf{F}}{\partial \Theta_k}\mathbf{P}(k) + \mathbf{Q}_1(k) \tag{121}$$

where:

$\Theta(k)$	Neural network weights at time k
$\mathbf{t}(k)$	Vector of desired outputs at time k
$\mathbf{F}(\Theta(k), \mathbf{p}(k))$	Nonlinear network input/output function at time k
$\frac{\partial \mathbf{F}}{\partial \Theta_k}$	Derivative of network outputs w.r.t. the weights
$\mathbf{Q}_1(k)$	Correlation matrix of training noise in weight vector
$\mathbf{Q}_2(k)$	Correlation matrix of noise in target vector
$\mathbf{K}(k)$	Kalman gain at time k
$\mathbf{P}(k+1)$	Correlation matrix of the error in $\Theta(k+1)$

Notice that the form of Eq. (120) is similar to all the neural network update equations presented thus far, with the new weights being the sum of the old weights and the product of an error and another term. The multiplying term $\mathbf{K}(k)$ involves a complete, recursively computed derivative matrix $\frac{\partial \mathbf{F}}{\partial \Theta_{(1\ldots k)}}$, and an error correlation matrix $\mathbf{P}(k+1)$. The rows of the derivative matrix $\frac{\partial \mathbf{F}}{\partial \Theta_{(1\ldots k)}}$ which are computed as $\frac{\partial \mathbf{F}}{\partial \Theta_k}$ at each iteration of the algorithm, may be computed using standard backpropagation for feedforward networks. (Dynamic backpropagation [16] must be used for recurrent networks.) The order of the inversion in Eq. (119) is equal to the number of network outputs. This algorithm is very similar to the recursive Gauss-Newton algorithm stated in Eq. (113) - Eq.

(116). In fact, it can be shown that the EKF and RGN algorithms are equivalent if the correlation matrices $\mathbf{Q}_1(k)$, and $\mathbf{Q}_2(k)$ are omitted from the EKF neural network training equations, and a weighted training cost function defined by Eq. (70) and Eq. (73) is used in the EKF development.

The numerical stability, convergence rate, and overall EKF algorithm performance are effected by the choices of $\mathbf{Q}_1(k)$, and $\mathbf{Q}_2(k)$. The $\mathbf{Q}_1(k)$ and $\mathbf{Q}_2(k)$ matrices are often "tweaked" for optimum performance on a problem by problem basis.

Chapter Summary

In this chapter several "on-line" methods of numerical optimization which may be used for training neural networks have been presented: the on-line steepest descent method, the recursive least-squares (RLS) algorithm, the recursive Gauss-Newton (RGN) method and the extended Kalman filter. The development of all these algorithms, with the exception of the extended Kalman filter, were derived by starting with batch processing methods and making the methods recursive using matrix algebra. Most notably, the *matrix inversion lemma* was employed to create recursive solutions to the least-squares algorithms. The EKF-based algorithm was developed for training multi-layer-perceptron networks by treating the weights of the network as states in an unforced nonlinear dynamical system, and applying the standard EKF algorithm. Examples showing the application of the steepest descent and RGN algorithms to training a simple nonlinear MLP network were presented. The example data showed "noisy" weight trajectories due to the incremental nature of these on-line methods.

Chapter 4

UNDERDETERMINED LINEARIZED LEAST-SQUARES TRAINING

Introduction

In this chapter an efficient "on-line" method of training complex neural networks which is based on the underdetermined linearized least squares solution will be presented. Henceforth this method will be referred to as the ULLS method. The development of a refinement to the ULLS algorithm which is a fully recursive algorithm requiring no matrix inversion is also presented. This algorithm will be referred to as the RULLS algorithm. A mathematical development and discussion of the algorithms will be presented in this chapter.

First-order, stochastic gradient descent methods can be used in training neural networks adaptively, but they often exhibit poor performance when used with complex (i.e. nonlinear, recurrent) network structures. Standard higher-order optimization methods, such as the Gauss-Newton method or the extended Kalman filter, are used to solve the linearized least-squares problem using the pseudo-inverse for the overdetermined case. These methods generally perform much better than gradient descent methods, but involve numerical operations on square matrices which are proportional in size to the number of parameters in the network. In a typical filtering or control problem the tapped-delay line [2] structure is used as the network input. If a fully connected neural network is used, the

tapped-delay can easily contain enough taps to necessitate having a large number of weights. If a system incorporates multiple tapped-delay lines, as we often see in neural network control applications [17], the number of weights required can be quite large and the associated computational burden can be very demanding if "on-line" training is required. The computational burden and memory requirement associated with training such a network can be prohibitive when implementing one of the standard higher-order optimization methods in a real-time system.

In this chapter a review of the linear least squares method for the overdetermined case and a more detailed development of linear least squares for the underdetermined case will be given. This development is then extended to the overdetermined and underdetermined cases for linearized least squares parameter estimation for nonlinear functions. The development of an incremental neural network training method which requires a matrix inversion operation is presented. Simulations in which the ULLS and RULLS methods are compared to both the on-line steepest descent and recursive linearized least-squares RLLS algorithms will be provided in Chapter 6.

Overdetermined Linear Least Squares Solution (Review)

In Chapter 2 a development of the linear least squares method was given. Recall that in this method the function to be minimized is the familiar sum-of-squared-errors function:

$$F(\Theta) = \sum_{j=1}^{N} e_j^2(\hat{\Theta}) = \mathbf{e}^T(\hat{\Theta})\mathbf{e}(\hat{\Theta}) \tag{122}$$

where:
$$\mathbf{e}(\hat{\Theta}) = [\mathbf{t} - \mathbf{a}(\hat{\Theta})] \tag{123}$$

In Eq. (122) the vector $\mathbf{e}(\hat{\Theta})$ represents the difference between a vector of desired outputs $\boldsymbol{t}$, and vector of outputs $\mathbf{a}(\hat{\Theta})$, produced by a linear function of the form,

$$a(n) = \hat{\theta}_1 g_1(n) + \hat{\theta}_2 g_2(n) + \ldots + \hat{\theta}_M g_M(n) \tag{124}$$

where $\hat{\theta}_1 \ldots \hat{\theta}_M$ are parameter estimates which form the vector $\hat{\Theta}$ and $g_1(n), g_2(n), \ldots, g_M(n)$ are scalar inputs. These scalar inputs can be nonlinear functions of other systems inputs. If we substitute Eq. (124) for all index values $(1 \ldots n \ldots N)$ into Eq. (123) and use Eq. (122), the following equation results:

$$F(\hat{\Theta}) = [\mathbf{t} - \mathbf{G}\hat{\Theta}]^T [\mathbf{t} - \mathbf{G}\hat{\Theta}] \tag{125}$$

where:

$$\mathbf{G} = \begin{bmatrix} g_1(1) & g_2(1) & \ldots & g_M(1) \\ g_1(2) & g_2(2) & \ldots & g_M(2) \\ \vdots & \vdots & \vdots & \vdots \\ g_1(N) & g_2(N) & \ldots & g_M(N) \end{bmatrix} \tag{126}$$

Eq. (125) can be rewritten as:

$$\nabla F(\hat{\Theta}) = -2[\mathbf{t}^T \mathbf{G}]^T + 2\mathbf{G}^T \mathbf{G}\hat{\Theta} \tag{127}$$

Taking the gradient of Eq. (127) with respect to Θ yields:

$$\nabla F(\hat{\Theta}) = -2[\mathbf{t}^T \mathbf{G}]^T + 2\mathbf{G}^T \mathbf{G}\hat{\Theta} \tag{128}$$

To find the parameters which minimize Eq. (128), we set the gradient equal to zero and solve for $\hat{\Theta}$. The resulting least squares estimate is

$$\hat{\Theta} = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{t} \qquad (129)$$

As shown in Chapter 3, the *Matrix Inversion Lemma* [3] can be used to develop an adaptive, recursive method of computing the solution to Eq. (129) which may be summarized as:

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + \mathbf{K}(k+1)[\mathbf{t}_k(k+1) - \mathbf{g}^T(k+1)\hat{\Theta}(k)] \qquad (130)$$

where

$$\mathbf{K}(k+1) = \mathbf{P}(k+1)\mathbf{g}(k+1)\gamma(k+1) \qquad (131)$$

and

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{P}(k)\mathbf{g}(k+1)[\mathbf{g}^T(k+1)\mathbf{P}(k)\mathbf{g}(k+1) + \gamma(k+1)^{-1}]^{-1}\mathbf{g}^T(k+1)\mathbf{P}(k) \qquad (132)$$

Remember from Chapter 3 that γ is a diagonal matrix with the forgetting factor γ on the diagonal which is necessary for adaptive training. The implementation of Eq. (130), Eq. (131) and Eq. (132) proceeds as follows:

$$\mathbf{P}(k) \rightarrow \mathbf{K}(k+1) \rightarrow \hat{\Theta}(k+1) \rightarrow \mathbf{P}(k+1) \qquad (133)$$

The solution of Eq. (129) or Eq. (130) is only valid when the number of measurements available exceeds the number of parameters to be estimated. As stated in Chapter 2, this situation is known as the *overdetermined* case [4].

Underdetermined Linear Least Squares Solution

Now let us suppose that there are fewer measurements available than there are parameters to be estimated. This situation can be described as the *underdetermined* case [4]. When we have this situation, Eq. (129) cannot be used. The under-constrained nature of this problem dictates that a single unique solution does not exist. To remedy this we must constrain the problem sufficiently as to force a unique solution. Looking at Eq. (125) we can see that if an ideal set of parameters exists then,

$$\mathbf{G}\hat{\Theta} = \boldsymbol{t} \tag{134}$$

and $\hat{\Theta}$ represents a perfect solution to Eq. (134). A straight-forward way of finding a suitable solution to the underdetermined problem is to minimize the sum of the squared parameters, while enforcing the following constraint:

$$\mathbf{G}\hat{\Theta} = \boldsymbol{t} \tag{135}$$

In other words, we want to perform the minimization with respect to $\hat{\Theta}$, and λ, where $\hat{\Theta}$ is the vector of parameters and λ is a vector of Lagrange multipliers [19].

$$\boldsymbol{min}\{\hat{\Theta}^T\hat{\Theta} + \lambda^T[\mathbf{G}\hat{\Theta} - \boldsymbol{t}]\} \tag{136}$$

Eq. (136) can be rewritten as:

$$\boldsymbol{min}\{\hat{\Theta}^T\hat{\Theta} + [\mathbf{G}\hat{\Theta} - \boldsymbol{t}]^T\lambda\} \tag{137}$$

Taking the gradient of Eq. (137) with respect to $\hat{\Theta}$ and λ and setting the result equal to zero yields:

$$\hat{\Theta} + \mathbf{G}^T\lambda = \mathbf{0} \tag{138}$$

and,

$$\mathbf{G}\hat{\Theta} - t = \mathbf{0} \tag{139}$$

Solving Eq. (138) for λ we get,

$$\lambda = -[\mathbf{G}\mathbf{G}^T]^{-1}\mathbf{G}\hat{\Theta} \tag{140}$$

but from Eq. (139) we know that:

$$\mathbf{G}\hat{\Theta} = t \tag{141}$$

Substituting Eq. (141) into Eq. (140) yields:

$$\lambda = -[\mathbf{G}\mathbf{G}^T]^{-1}t \tag{142}$$

Eq. (142) can now be plugged into Eq. (138) and the result can be solved for $\hat{\Theta}$ to show that:

$$\hat{\Theta} = \mathbf{G}^T[\mathbf{G}\mathbf{G}^T]^{-1}t \tag{143}$$

Eq. (143) is the solution to the underdetermined linear least squares problem. Notice that the matrix product which appears inside the inversion of Eq. (143) is an M x M matrix where M is the number of sample contributions to the $\mathbf{G}$ matrix. The outer product form of Eq. (143) does not lend itself to the insertion of a traditional exponential weighting

matrix to effect "forgetting" in adaptive applications. This structure also precludes the use of the standard matrix inversion lemma for efficient recursive computation of $[\mathbf{G}\mathbf{G}^T]^{-1}$. These mathematical difficulties must be overcome in order to develop a useful and efficient adaptive training algorithm. However, if a small window of sample contributions is used in forming the $\mathbf{G}$ matrix, the size of the resulting matrix to be inverted in Eq. (143) is small. This opens the possibility of computing parameter updates in "real-time" with fewer computations than with the standard RLS algorithm for the overdetermined case. In the following sections it will be shown that this type of algorithm can be extended for use with nonlinear functions by linearizing these systems prior to applying the algorithm. The solution to the underdetermined case when applied to linearized nonlinear functions is the basis for the neural network training method presented in this chapter.

Nonlinear Least-Squares for the Overdetermined Case (Review)

In Chapter 2 it was shown that the least-squares method may be used in minimizing nonlinear functions by first linearizing the function about a nominal set of parameter values, then applying the standard least-squares equations to calculate an incremental parameter adjustment. This process is repeated until a specified convergence criteria has been satisfied. Repeated here for convenience, the equations which describe the algorithm are:

$$\delta\hat{\Theta}_n = -[\mathbf{J}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)]^{-1}\mathbf{J}(\hat{\Theta}_n)^T\mathbf{e}(\hat{\Theta}_n) \tag{144}$$

$$\hat{\Theta}_{n+1} = \hat{\Theta}_n + \delta\hat{\Theta}_n \tag{145}$$

or,

$$\hat{\Theta}_{n+1} = \hat{\Theta}_n - [\mathbf{J}(\hat{\Theta}_n)^T\mathbf{J}(\hat{\Theta}_n)]^{-1}\mathbf{J}(\hat{\Theta}_n)^T\mathbf{e}(\hat{\Theta}_n) \tag{146}$$

where,

$\hat{\Theta}$ is a vector of unknown function parameters

$\boldsymbol{a}(\hat{\Theta})$ is a vector of outputs of a nonlinear function

$\mathbf{e}(\hat{\Theta})$ is a vector of errors between desired outputs $\boldsymbol{t}$, and function outputs $\boldsymbol{a}(\hat{\Theta})$

$\mathbf{J}(\hat{\Theta}_n)$ is the derivative of the function outputs w.r.t. the function parameters

$\delta\hat{\Theta}_n$ is the vector of incremental adjustments of the estimated parameters

In summary, the overdetermined least-squares solution to the linearized system is computed about the current set of parameter estimates to find the appropriate vector of parameter adjustments which are then used to compute an updated vector of parameter estimates.

Nonlinear Least-Squares for the Underdetermined Case

A linearized least squares solution can also be obtained for underdetermined nonlinear optimization problems. As was demonstrated in Chapter 2 for the overdetermined case, linear perturbation equations can be obtained by approximating the nonlinear function using a first-order Taylor series expansion. The same mathematical method previously described in this chapter for the underdetermined linear least squares problem can be applied to the linear perturbation equations to develop a linearized least squares solution for the underdetermined case. We start with Eq. (61) (Chapter 2) rewritten here for convenience, which results from substituting the first-order Taylor series approximation of the nonlinear function $\mathbf{a}(\hat{\Theta}_n)$ into the squared errors function.

$$F(\hat{\Theta}) \approx [\mathbf{e}(\hat{\Theta}_n) + \mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n]^T[\mathbf{e}(\hat{\Theta}_n) + \mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n] \tag{147}$$

Looking at Eq. (147) we can see that if an ideal a set of parameter adjustments exists then,

$$\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n = -\mathbf{e}(\hat{\Theta}_n) \tag{148}$$

and $\delta\hat{\Theta}_n$ represents an exact solution to Eq. (148). Following a method similar to that used in finding a solution to the underdetermined linear least squares case we minimize the sum of the squared parameter adjustments, while enforcing the following constraint:

$$\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n = -\mathbf{e}(\hat{\Theta}_n) \tag{149}$$

In other words, we want to perform the minimization with respect to $\hat{\Theta}$ and $\boldsymbol{\lambda}$, where $\hat{\Theta}$ is the vector of parameters and $\boldsymbol{\lambda}$ is a vector of Lagrange multipliers [19].

This gives us:

$$\boldsymbol{min}\{\delta\hat{\Theta}_n^T\delta\hat{\Theta}_n + \boldsymbol{\lambda}^T[\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n + \mathbf{e}(\hat{\Theta}_n)]\} \tag{150}$$

Eq. (150) can be rewritten as:

$$\boldsymbol{min}\{\delta\hat{\Theta}_n^T\delta\hat{\Theta}_n + [\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n + \mathbf{e}(\hat{\Theta}_n)]^T\boldsymbol{\lambda}\} \tag{151}$$

Taking the gradient of Eq. (151) with respect to $\delta\hat{\Theta}_n$ and $\boldsymbol{\lambda}$ respectively and setting the result equal to zero yields:

$$\delta\hat{\Theta}_n + \mathbf{J}(\hat{\Theta}_n)^T\boldsymbol{\lambda} = \mathbf{0} \tag{152}$$

and,

$$\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n + \mathbf{e}(\hat{\Theta}_n) = \mathbf{0} \tag{153}$$

Solving Eq. (152) for $\boldsymbol{\lambda}$ we get,

$$\boldsymbol{\lambda} = [\mathbf{J}(\hat{\Theta}_n)\mathbf{J}(\hat{\Theta}_n)^T]^{-1}\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n \tag{154}$$

but from Eq. (153) we know that:

$$\mathbf{J}(\hat{\Theta}_n)\delta\hat{\Theta}_n = -\mathbf{e}(\hat{\Theta}_n) \tag{155}$$

Substituting Eq. (155) into Eq. (154) yields:

$$\boldsymbol{\lambda} = -[\mathbf{J}(\hat{\Theta}_n)\mathbf{J}(\hat{\Theta}_n)^T]^{-1}\mathbf{e}(\hat{\Theta}_n) \tag{156}$$

This can now be plugged into Eq. (152) and the result can be solved for $\hat{\Theta}$ to show that:

$$\delta\hat{\Theta}_n = \mathbf{J}(\hat{\Theta}_n)^T[\mathbf{J}(\hat{\Theta}_n)\mathbf{J}(\hat{\Theta}_n)^T]^{-1}\mathbf{e}(\hat{\Theta}_n) \tag{157}$$

This result provides the incremental parameter update which can be used with the standard update equations to complete the underdetermined linearized least squares optimization algorithm which is:

$$\hat{\Theta}_{n+1} = \hat{\Theta}_n + \delta\hat{\Theta}_n \tag{158}$$

$$\hat{\Theta}_{n+1} = \hat{\Theta}_n - \mathbf{J}(\hat{\Theta}_n)^T[\mathbf{J}(\hat{\Theta}_n)\mathbf{J}(\hat{\Theta}_n)^T]^{-1}\mathbf{e}(\hat{\Theta}_n) \tag{159}$$

As with the underdetermined linear least squares method, the size of the matrix which must be inverted in Eq. (159) is dependent not on the number of parameters, but on

the number of measurements being considered. This is an interesting result. It means that the method can potentially be used to train neural networks having large numbers of weights, while only performing computations involving relatively small matrices, vectors and memory blocks. This method will be applied in the next section to the training of non-linear multi-layer neural networks.

Neural Network Training Using the ULLS Method with Matrix Inversion

In this section a method of using the underdetermined linearized least squares (ULLS) approach to incrementally train nonlinear neural networks will be presented. The method relies on applying Eq. (159), where the parameters $\hat{\Theta}$ are the network weights and the gradient matrix $\mathbf{J}(\hat{\Theta}_n)$ is a windowed Jacobian matrix $\mathbf{J}_{ULLS}$, for the network. The equation for a single update of the network weights can be written as:

$$\Theta_{n+1} = \Theta_n - \alpha[\mathbf{J}_{ULLS_n}{}^T[\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}{}^T]^{-1}\mathbf{e}_n] \tag{160}$$

A slight modification to Eq. (159) appears in Eq. (160) with the addition of α, a small, positive scalar ($0 < \alpha \leq 1$) which multiplies the weight update. This additional term can be adjusted to help stabilize the algorithm in much the same way it is used in the so-called Damped Recursive Gauss-Newton algorithm [20].

The partial Jacobian matrix used in Eq. (160) can be assembled in many ways. A simple method is to simply process "blocks" of the full Jacobian matrix which contain fewer rows than columns so as to preserve the underdetermined state as each consecutive block of data is processed. Each time a block is processed the weights are updated. Another method of constructing the partial Jacobian matrix is to use a "moving window" of the full

Jacobian. The length of the window is set such that the number of rows in the Jacobian matrix window is less than the number of columns. As a new row of the Jacobian matrix becomes available, it is added to the window, and the oldest row in the window is removed. In this method the data is updated at each index in time to reflect a window of past measurements. This update can be described by Eq. (161) below, where j_n is the most recently computed row in the Jacobian matrix.

$$\mathbf{J}_{ULLS_n} = \begin{bmatrix} \leftarrow & \mathrm{j}_{n-M} & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & \mathrm{j}_{n-1} & \rightarrow \\ \leftarrow & \mathrm{j}_n & \rightarrow \end{bmatrix} \tag{161}$$

Both of the methods described for assembling the partial Jacobian matrix cause the algorithm to "forget" the effects of older measurements, making the method suitable for adaptive network training. As stated previously, the required matrix inversion in Eq. (160) is of order M, which is the number of measurements considered in the partial Jacobian matrix. This is of particular importance in training neural networks.

As stated previously, a fully-connected neural network can easily contain a large number of weights. This is especially true in applications in which tapped-delay inputs are used. A large number of weights can be required to accommodate the inputs from even a modest length tapped-delay line. Unfortunately, tapped-delay lines are commonly used in real-time operations where conventional recursive linearized least squares training of networks containing a large number of weights is impractical because of computational requirements. These methods do not require direct inversion of large matrices, but do require

multiplication and addition operations on matrices which are of order N, the number of network parameters. The ULLS method of training networks requires a direct matrix inversion of order M, the number of measurements to be considered in the Jacobian matrix window. If an efficient method such as Gaussian Elimination [19] is used to compute the required inverse in Eq. (160) the number of floating point operations required for each sample is

$$\frac{flops_{ULLS}}{sample} = \frac{2}{3}m^3 + 3m^2 n - \frac{1}{2}m^2 + n + m \tag{162}$$

In Eq. (162), m is the number of terms in the Jacobian matrix window, and n is the number of parameters. This result indicates that it may be most appropriate to use relatively few terms in the Jacobian matrix window when using the ULLS algorithm.

If we consider the case where only one term is considered in the Jacobian window, we can see that this is essentially gradient descent with a variable learning rate. Consider Eq. (160). If only one row is present in the Jacobian matrix window, the matrix product $\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T$ becomes a vector product, which is simply a scalar. This scalar is multiplied by $\mathbf{J}_{ULLS_n}^T$, which for the one term case is simply the gradient vector for one error with respect to the weights. Therefore, the vector product $\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T$ simply acts as a varying scalar which multiplies the gradient in exactly the same way the learning rate α does.

Recursive Underdetermined Linearized Least Squares (RULLS) Training Algorithm

In this section, a modification to the ULLS algorithm which utilizes an exact, but computationally efficient recursive method of computing the required inverse of

$\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T$ will be presented. Henceforth, the method will be referred to as the RULLS algorithm.

If we recall Eq. (160), rewritten here for convenience, we remember that a matrix inversion is required when computing a weight update:

$$\Theta_{n+1} = \Theta_n - \alpha[\mathbf{J}_{ULLS_n}^T[\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T]^{-1} e_n] \tag{163}$$

The development of a recursive algorithm for the computation of the inverse of $\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T$ will allow for a fully recursive implementation of Eq. (163). For this development a "moving window" of the Jacobian matrix will be assumed to incorporate a "forgetting" mechanism as described previously. As a new row of the Jacobian matrix becomes available, it is added to the window and the oldest row in the window is removed. In terms of the incremental $\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T$ matrix, the requirement is to compute the inverse of a square matrix moving down the diagonal of the $\mathbf{J}_n\mathbf{J}_n^T$ matrix which accounts for all time steps up to n. The smaller $\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T$ matrix is of order M where M is the number of samples to be considered in the underdetermined weight update at time n. Eq. (164) shows how the incremental $\mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T$ matrix is positioned within the $\mathbf{J}_n\mathbf{J}_n^T$ matrix.

$$\mathbf{H}_n = \mathbf{J}_n\mathbf{J}_n^T = \begin{bmatrix} h_{(1,1)} & \cdots & h_{(1,n-M)} & \cdots & h_{(1,n)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{(n-M,1)} & \cdots & h_{(n-M,n-M)} & \cdots & h_{(n-M,n)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ h_{(n,1)} & \cdots & h_{(n,n-M)} & \cdots & h_{(n,n)} \\ & & \downarrow & & \searrow \end{bmatrix} \rightarrow \mathbf{J}_{ULLS_n}\mathbf{J}_{ULLS_n}^T = \mathbf{H}_{ULLS_n} \tag{164}$$

At time $n+1$, the $\mathbf{J}_{ULLS}\mathbf{J}_{ULLS}^T$ matrix, or equivalently the $\mathbf{H}_{ULLS}$ matrix will be formed by moving the inner brackets shown in Eq. (164) down one position and to the right one position within the $\mathbf{H}_n$ matrix. In order to evaluate Eq. (160), $\mathbf{H}_{ULLS}^{-1}$ must be evaluated at each time step. When using the standard ULLS algorithm a standard full matrix inversion is performed to evaluate $\mathbf{H}_{ULLS}^{-1}$. As stated previously, the goal of the RULLS algorithm is to evaluate $\mathbf{H}_{ULLS}^{-1}$ at each time step without performing a standard matrix inversion. In order to accomplish this task, the *Block Matrix Inversion Lemma* [25] is employed. A statement of the block matrix inversion lemma follows.

Block Matrix Inversion Lemma

If the matrices **A**, **B**, **C**, *and* **D** *are respectively, an n x n, an n x m, an m x n, and an m x m matrix, then*

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}(\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}(\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \\ -(\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1} & (\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \end{bmatrix} \tag{165}$$

provided $|\mathbf{A}| \neq 0$ *and* $|\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B}| \neq 0$, *and*

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} (\mathbf{A}-\mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & -(\mathbf{A}-\mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}(\mathbf{A}-\mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & \mathbf{D}^{-1}\mathbf{C}(\mathbf{A}-\mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}\mathbf{B}\mathbf{D}^{-1}+\mathbf{D}^{-1} \end{bmatrix} \quad (166)$$

provided $|\mathbf{D}| \neq 0$ *and* $|\mathbf{A}-\mathbf{B}\mathbf{D}^{-1}\mathbf{C}| \neq 0$.

A detailed proof of the block matrix inversion lemma may be found in [25]. Fun [26] shows a general method for order update and order downdate which is based on the block matrix inversion lemma. Fun's method is the basis of derivation of the RULLS method which follows.

The RULLS algorithm requires that the $\mathbf{H}_{ULLS_n}$ matrix and its inverse be "updated" by one order to include the newest row of the Jacobian matrix, then "down-dated" by one order to remove the oldest row in the Jacobian matrix window. The updated ULLS Jacobian matrix $\mathbf{J}_{up_{n+1}}$, which contains one more row than $\mathbf{J}_{ULLS_n}$, can be expressed as:

$$\mathbf{J}_{up_{n+1}} = \begin{bmatrix} \mathbf{J}_{ULLS_n} \\ --- \\ \mathbf{j}_{n+1} \end{bmatrix} \quad (167)$$

Now define the matrix $\mathbf{V}_{n+1}$ which is one order larger than the $\mathbf{H}_{ULLS_n}$ matrix as:

$$\mathbf{V}_{n+1} = \mathbf{J}_{up_{n+1}}\mathbf{J}_{up_{n+1}}^T = \begin{bmatrix} \mathbf{H}_{ULLS_n} & \mathbf{J}_{ULLS_n}\mathbf{j}_{n+1}^T \\ \vdots & \vdots \\ \mathbf{j}_{n+1}\mathbf{J}_{ULLS_n}^T & \mathbf{j}_{n+1}\mathbf{j}_{n+1}^T \end{bmatrix} \quad (168)$$

Eq. (165) may be used to compute $\mathbf{V}_{n+1}^{-1}$ given $\mathbf{H}_{ULLS_n}^{-1}$ from the previous time step by letting $\mathbf{A}^{-1}$ equal $\mathbf{H}_{ULLS_n}^{-1}$, $\mathbf{B}$ equal $\mathbf{J}_{ULLS_n}\mathbf{j}_{n+1}^T$, $\mathbf{C}$ equal $\mathbf{j}_{n+1}\mathbf{J}_{ULLS_n}^T$ and $\mathbf{D}$ equal $\mathbf{j}_{n+1}\mathbf{j}_{n+1}^T$. The result is:

$$\mathbf{V}_{n+1}^{-1} = \begin{bmatrix} \mathbf{H}_{ULLS_n}^{-1} + \mathbf{H}_{ULLS_n}^{-1}\mathbf{J}_{ULLS_n}\mathbf{j}_{n+1}^T \Upsilon \mathbf{j}_{n+1}\mathbf{J}_{ULLS_n}^T\mathbf{H}_{ULLS_n}^{-1} & -\mathbf{H}_{ULLS_n}^{-1}\mathbf{J}_{ULLS_n}\mathbf{j}_{n+1}^T\Upsilon \\ -\Upsilon\mathbf{j}_{n+1}\mathbf{J}_{ULLS_n}^T\mathbf{H}_{ULLS_n}^{-1} & \Upsilon \end{bmatrix} \tag{169}$$

where

$$\Upsilon = (\mathbf{j}_{n+1}\mathbf{j}_{n+1}^T - \mathbf{j}_{n+1}\mathbf{J}_{ULLS_n}^T\mathbf{H}_{ULLS_n}^{-1}\mathbf{J}_{ULLS_n}\mathbf{j}_{n+1}^T)^{-1} \tag{170}$$

is a scalar quantity.

Eq. (169) results in the inverse of a portion of the $\mathbf{JJ}^T$ matrix at time $n+1$ which is one order larger than desired and still reflects the unwanted presence of the oldest row of the windowed Jacobian matrix. Now the contribution of the oldest row of the $\mathbf{J}_{up_{n+1}}$ matrix must be removed in the computation of the desired $\mathbf{H}_{ULLS_{n+1}}^{-1}$ matrix which is required for updating the weights at time $n+1$.

The derivation of a "down-date" algorithm begins again with the block matrix inversion lemma, but in the form shown in Eq. (166), repeated here for convenience.

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} (\mathbf{A}-\mathbf{BD}^{-1}\mathbf{C})^{-1} & -(\mathbf{A}-\mathbf{BD}^{-1}\mathbf{C})^{-1}\mathbf{BD}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}(\mathbf{A}-\mathbf{BD}^{-1}\mathbf{C})^{-1} & \mathbf{D}^{-1}\mathbf{C}(\mathbf{A}-\mathbf{BD}^{-1}\mathbf{C})^{-1}\mathbf{BD}^{-1}+\mathbf{D}^{-1} \end{bmatrix} \tag{171}$$

Eq. (171) can be expressed as

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{\Psi} & -\mathbf{\Psi}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}\mathbf{\Psi} & \mathbf{D}^{-1}\mathbf{C}\mathbf{\Psi}\mathbf{B}\mathbf{D}^{-1} + \mathbf{D}^{-1} \end{bmatrix} \tag{172}$$

where,

$$\mathbf{\Psi} = (\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} \tag{173}$$

Eq. (172) can be expressed as

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{\Psi} & -\mathbf{\Psi}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}\mathbf{\Psi} & \mathbf{D}^{-1}\mathbf{C}\mathbf{\Psi}\mathbf{B}\mathbf{D}^{-1} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}^{-1} \end{bmatrix} \tag{174}$$

Eq. (174) can be rearranged to obtain

$$\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}^{-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} - \begin{bmatrix} \mathbf{\Psi} & -\mathbf{\Psi}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}\mathbf{\Psi} & \mathbf{D}^{-1}\mathbf{C}\mathbf{\Psi}\mathbf{B}\mathbf{D}^{-1} \end{bmatrix} \tag{175}$$

If $\mathbf{\Psi}$ is a scalar, Eq. (175) can be expressed as

$$\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}^{-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} - \begin{bmatrix} \psi \\ -\mathbf{D}^{-1}\mathbf{C}\,\psi \end{bmatrix} \begin{bmatrix} \psi & -\psi\mathbf{B}\mathbf{D}^{-1} \end{bmatrix} / \psi \tag{176}$$

From Eq. (172) we see that $\begin{bmatrix} \psi \\ -\mathbf{D}^{-1}\mathbf{C}\,\psi \end{bmatrix}$ is simply the first column of $\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1}$, $\begin{bmatrix} \psi & -\psi\mathbf{B}\mathbf{D}^{-1} \end{bmatrix}$ is the first row of $\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1}$ and ψ is the upper left element of $\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1}$. This makes it possible to compute $\mathbf{D}^{-1}$ using only elements of $\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1}$.

Eq. (176) can now be used to compute $\mathbf{H}_{ULLS_{n+1}}^{-1}$ by letting $\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1}$ equal $\mathbf{V}_{n+1}^{-1}$

from Eq. (169). Plugging the appropriate values into Eq. (176) yields

$$\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_{ULLS_{n+1}}^{-1} \end{bmatrix} = \mathbf{V}_{n+1}^{-1} - \begin{bmatrix} v_{n+1(1,1)}^{-1} \\ \\ v_{n+1(M+1,1)}^{-1} \end{bmatrix} \begin{bmatrix} v_{n+1(1,1)}^{-1} & \dots & v_{n+1(1,M+1)}^{-1} \end{bmatrix} / v_{n+1(1,1)}^{-1} \tag{177}$$

A combination of Eq. (160), Eq. (169), Eq. (170) and Eq. (177) comprise the fully recursive RULLS algorithm which is

$$\Upsilon_n = [\mathbf{j}_n \mathbf{j}_n^T - \mathbf{j}_n \mathbf{J}_{ULLS_{n-1}}^T \mathbf{H}_{ULLS_{n-1}}^{-1} \mathbf{J}_{ULLS_n} \mathbf{j}_{n+1}^T]^{-1} \tag{178}$$

$$\mathbf{V}_n^{-1} = \begin{bmatrix} \mathbf{H}_{ULLS_{n-1}}^{-1} + \mathbf{H}_{ULLS_{n-1}}^{-1} \mathbf{J}_{ULLS_{n-1}} \mathbf{j}_{n1}^T \Upsilon_n \mathbf{j}_n \mathbf{J}_{ULLS_{n-1}}^T \mathbf{H}_{ULLS_{n-1}}^{-1} & -\mathbf{H}_{ULLS_{n-1}}^{-1} \mathbf{J}_{ULLS_{n-1}} \mathbf{j}_n^T \Upsilon_n \\ -\Upsilon_n \mathbf{j}_n \mathbf{J}_{ULLS_{n-1}}^T \mathbf{H}_{ULLS_{n-1}}^{-1} & \Upsilon_n \end{bmatrix} \tag{179}$$

$$\begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_{ULLS_n}^{-1} \end{bmatrix} = \mathbf{V}_n^{-1} - \begin{bmatrix} v_n^{-1}{}_{(1,1)} \\ \\ v_n^{-1}{}_{(M+1,1)} \end{bmatrix} \begin{bmatrix} v_n^{-1}{}_{(1,1)} & \dots & v_n^{-1}{}_{(1,M+1)} \end{bmatrix} / v_n^{-1}{}_{(1,1)} \tag{180}$$

$$\mathbf{W}_{n+1} = \mathbf{W}_n - \alpha[\mathbf{J}_{ULLS_n}{}^T [\mathbf{H}_{ULLS_n}]^{-1} e_n] \tag{181}$$

The computational requirement associated with each part of the RULLS algorithm is shown in Table 1. In Table 1, m is the number of terms in the Jacobian matrix window, and n is the number of parameters. The level of computation required for the RULLS algorithm is approximately an order less than is required for the ULLS algorithm. Figure 34 shows the impact the number of parameters and the length of the Jacobian matrix window has on the number of floating point operations required for both the ULLS and RULLS methods.

RULLS Algorithm Element	**Computational Burden** *m = number of rows in* $\mathbf{J}_{\mathrm{ULLS}}$ *n = number of parameters*
Υ_n	$2m^2 + 2nm + 2n$
$\mathbf{V}_n^{-1}$	$3m^2 + 2nm + 2n + 2m$
$\mathbf{H}_{ULLS_n}^{-1}$	$m^2 - 2m$
$\mathbf{W}_{n+1}$	$2m^2 + 2nm - m - n$
Total	$8m^2 + 6nm - m + n$

Table 1 Computational requirements for the RULLS algorithm

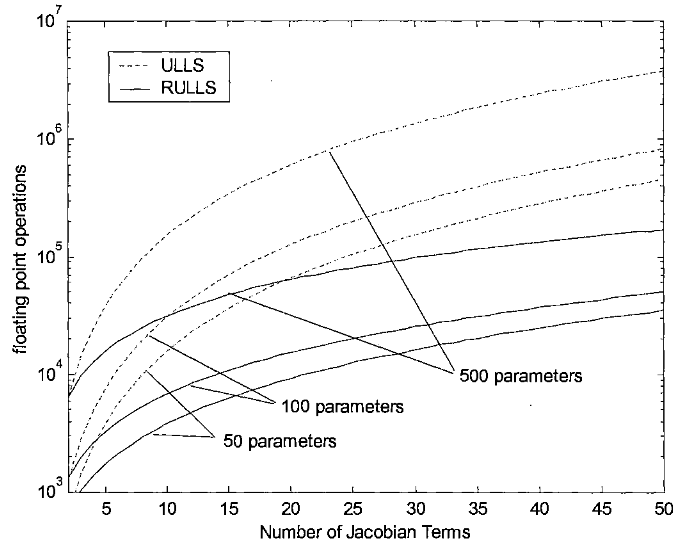


Figure 34 Computation Requirement Comparison for ULLS and RULLS Algorithms

Numerical Stability of the RULLS Algorithm

The RULLS algorithm was implemented using an INTEL Pentium 3 processor and MATLAB programming software. Simulations were run to test the RULLS algorithm against the standard ULLS algorithm on identical problems. These simulations revealed that the RULLS algorithm suffers from numerical stability problems due to the finite precision of the computer. The inverse computation required by the ULLS algorithm was compared to the recursive inverse computation performed using the RULLS algorithm. These results showed very similar inverse matrix values in the early iterations of both algorithms. As more and more iterations were performed, the inverse matrix values computed using the RULLS algorithm slowly diverged from the true inverse matrix values appearing in the ULLS computation. The values computed by the RULLS algorithm did not rapidly "blow-up", but became incorrect enough to produce very poor training results. Future research work could be performed to focus on a numerically stable RULLS algorithm.

Chapter Summary

A method for training neural networks on-line, using the solution to the underdetermined linearized least-squares problem, has been presented. A mathematical development of the ULLS algorithm was given. The ULLS algorithm was then further refined to produce the RULLS algorithm which is mathematically equivalent to the ULLS method, but is recursive in nature, requiring no direct matrix inversion. The method shows good promise for application in real-time systems because it allows weight updates to be made using fewer measurements than weights in the network. It was shown that the method reduces to

gradient descent with variable learning rate if only one Jacobian term is considered. Another variation on the algorithm which might be considered is to adaptively determine the optimal trade-off between the number of Jacobian terms considered and the update rate (determined by overlap in the partial Jacobian windows) to obtain the highest performance for a given computational capability. A comparison between the ULLS and RULLS methods has shown that many fewer floating point operations are required for the RULLS method than for the ULLS method. As we will see in Chapter 6, the number of measurements considered for each weight update does have an impact on the algorithm performance as well as the computational burden. The performance of the ULLS and RULLS algorithms are examined and compared to other on-line training methods in Chapter 6.

Chapter 5

INCREMENTAL LEVENBERG-MARQUARDT OPTIMIZATION

Introduction

The Levenberg-Marquardt (LM) optimization algorithm has been proven to be one of the most effective and useful methods of training neural networks. Because of the success of the LM method in neural network applications where batch processing is appropriate, it is natural to investigate the possibility of developing a recursive LM algorithm for real-time applications. Interest in developing a recursive LM algorithm is evidenced by recent work by Stan and Kamen [21] on a block recursive LM algorithm. Ngia and Sjoberg [22] have also recently published work on an approximate recursive LM algorithm. In this chapter two new approaches to the development of a recursive LM algorithm will be presented. The goal of both methods is to allow for parameter updates at each time step such that if the same block of data is processed using the batch LM algorithm a similar result will be obtained. A general development of the recursive LM equations will be presented first. This development will be the starting point for both of the new recursive LM algorithms, which only differ in the way in which matrix inversion is avoided in evaluating the recursive solution. The first method to be presented is the Recursive Levenberg-Marquardt with Matrix Inversion Lemma (RLMMIL) method. A short section showing some simulation results will follow the RLMMIL development section. The second method to be presented

is the Recursive Levenberg-Marquardt with Caley-Hamilton Inverse Approximation (RLMCH) method. Simulation results for this method will also be presented in a short section following the mathematical development of the RLMCH method. Conclusions will be provided in a final section.

Recursive Levenberg-Marquardt Equations

The description of the LM method presented in Chapter 2 showed that the method is a modification of the batch Gauss-Newton Method. The basic Gauss-Newton batch update equation given in Chapter 2 is:

$$\Theta_{n+1} = \Theta_n - [\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)]^{-1}\mathbf{J}^T(\Theta)\mathbf{e}(\Theta) \tag{182}$$

In Chapter 3 it was shown that an incremental solution to Eq. (182) can be developed by using the *Matrix Inversion Lemma* [3] to avoid having to perform a matrix inversion. A restatement of the matrix inversion lemma follows.

Matrix Inversion Lemma

If the matrices **A**, **B**, **C**, *and* **D** *satisfy the equation*

$$\mathbf{B}^{-1} = \mathbf{A}^{-1} + \mathbf{C}^T\mathbf{D}^{-1}\mathbf{C} \tag{183}$$

where all matrix inverses are assumed to exist, then

$$\mathbf{B} = \mathbf{A} - \mathbf{A}\mathbf{C}^T(\mathbf{C}\mathbf{A}\mathbf{C}^T + \mathbf{D})^{-1}\mathbf{C}\mathbf{A} \tag{184}$$

A detailed proof of the matrix inversion lemma may be found in [3]. It was shown in Chapter 3 that the $\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)$ term in Eq. (182) can be updated for each new data point by adding an outer product matrix which is produced by multiplying the transpose of the new row

of the Jacobian matrix by itself. This is the form represented by Eq. (183) and thus Eq. (184) can be used to recursively compute the inverse of $\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)$.

If a diagonal matrix is added to $\mathbf{J}^T(\Theta)\mathbf{J}(\Theta)$ in Eq. (182) the result is the Levenberg-Marquardt algorithm. The Levenberg-Marquardt batch update equation given in Chapter 2 is:

$$\Theta_{n+1} = \Theta_n - [\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + \mu\mathbf{I}]^{-1}\mathbf{J}^T(\Theta)\mathbf{e}(\Theta) \tag{185}$$

To begin the development of the recursive Levenberg-Marquardt update equations, the weight update at time n can be defined as:

$$\Delta\Theta_n = [\mathbf{J}_n^T(\Theta)\mathbf{J}_n(\Theta) + \mu_n\mathbf{I}]^{-1}\mathbf{J}_n^T(\Theta)\mathbf{e}_n(\Theta) \tag{186}$$

Eq. (186) can be rewritten as:

$$\Delta\Theta_n = \mathbf{P}_n\mathbf{J}_n^T(\Theta)\mathbf{e}_n(\Theta) \tag{187}$$

where,

$$\mathbf{P}_n = [\mathbf{J}_n^T(\Theta)\mathbf{J}_n(\Theta) + \mu_n\mathbf{I}]^{-1} \tag{188}$$

and,

$$\mathbf{P}_{n+1} = [\mathbf{J}_{n+1}^T(\Theta)\mathbf{J}_{n+1}(\Theta) + \mu_{n+1}\mathbf{I}]^{-1} \tag{189}$$

Additionally, taking the inverse of both sides of Eq. (188) and Eq. (189) we have:

$$\mathbf{P}_n^{-1} = [\mathbf{J}_n^T(\Theta)\mathbf{J}_n(\Theta) + \mu_n\mathbf{I}] \tag{190}$$

and,

$$\mathbf{P}_{n+1}^{-1} = [\mathbf{J}_{n+1}^T(\Theta)\mathbf{J}_{n+1}(\Theta) + \mu_{n+1}\mathbf{I}] \tag{191}$$

The Jacobian matrix at time $n + 1$ is simply the Jacobian matrix at time n with one additional row appended to account for the next time step at $n + 1$. Because of this the following equation can be written.

$$\mathbf{J}_{n+1}{}^{T}(\Theta)\mathbf{J}_{n+1}(\Theta) = [\mathbf{J}_{n}{}^{T}(\Theta)\mathbf{J}_{n}(\Theta) + \mathbf{j}_{n+1}^{T}\mathbf{j}_{n+1}] \tag{192}$$

In Eq. (192) $\mathbf{j}_{n+1}$ is the new row of the Jacobian matrix associated with time $n + 1$. The diagonal matrix of Eq. (191) can be expressed at time $n + 1$ as:

$$\mu_{n+1}\mathbf{I} = \mu_{n}\mathbf{I} + \Delta\mu_{n+1}\mathbf{I} \tag{193}$$

Inserting Eq. (192) and Eq. (193) into Eq. (191) yields:

$$\mathbf{P}_{n+1}^{-1} = \mathbf{J}_{n}{}^{T}(\Theta)\mathbf{J}_{n}(\Theta) + \mu_{n}\mathbf{I} + \mathbf{j}_{n+1}^{T}\mathbf{j}_{n+1} + \Delta\mu_{n+1}\mathbf{I} \tag{194}$$

Using Eq. (190), Eq. (194) can be rewritten as:

$$\mathbf{P}_{n+1}^{-1} = \mathbf{P}_{n}^{-1} + \mathbf{j}_{n+1}^{T}\mathbf{j}_{n+1} + \Delta\mu_{n+1}\mathbf{I} \tag{195}$$

To complete the development of the recursive LM algorithm Eq. (187) can be rearranged to produce:

$$\mathbf{J}_{n}{}^{T}(\Theta)\mathbf{e}_{n}(\Theta) = \mathbf{P}_{n}^{-1}\Delta\Theta_{n} \tag{196}$$

and,

$$\Delta\Theta_{n+1} = \mathbf{P}_{n+1}\mathbf{J}_{n+1}{}^{T}(\Theta)\mathbf{e}_{n+1}(\Theta) \tag{197}$$

but,

$$\mathbf{J}_{n+1}{}^{T}(\Theta)\mathbf{e}_{n+1}(\Theta) = \mathbf{J}_{n}{}^{T}(\Theta)\mathbf{e}_{n}(\Theta) + \mathbf{j}_{n+1}^{T}e_{n+1}(\Theta) \tag{198}$$

Substituting Eq. (196) into Eq. (198) and substituting the result into Eq. (197) yields:

$$\Delta\Theta_{n+1} = \mathbf{P}_{n+1}[\mathbf{P}_n^{-1}\Delta\Theta_n + \mathbf{j}_{n+1}^T e_{n+1}(\Theta)] \tag{199}$$

where, from Eq. (195)

$$\mathbf{P}_{n+1}^{-1} = \mathbf{P}_n^{-1} + \mathbf{j}_{n+1}^T\mathbf{j}_{n+1} + \Delta\mu_{n+1}\mathbf{I} \tag{200}$$

Eq. (199) and Eq. (200) can be used with the standard update equation given below,

$$\Theta_{n+1} = \Theta_n - \Delta\Theta_{n+1} \tag{201}$$

to recursively compute Levenberg-Marquardt parameter updates.

The LM parameter update algorithm formed by Eq. (199), Eq. (200) and Eq. (201) is of little use because a $dim(\Theta)$ inversion must be performed to obtain $\mathbf{P}_{n+1}$ in Eq. (199). The addition of the diagonal matrix inside the inversion in Eq. (185) makes recursive computation of $[\mathbf{J}^T(\Theta)\mathbf{J}(\Theta) + \mu\mathbf{I}]^{-1}$ by direct application of the matrix inversion lemma impossible. The problem arises in Eq. (200) where the $\Delta\mu_{n+1}\mathbf{I}$ matrix precludes the use of the matrix inversion lemma for a one-step recursive computation of $\mathbf{P}_{n+1}$.

Eq. (199), Eq. (200) and Eq. (201) represent a starting point for the development of the two recursive Levenberg-Marquardt algorithms which will be presented in the following sections.

Recursive Levenberg-Marquardt Using Matrix Inversion Lemma

In this section a new method of evaluating the recursive LM algorithm given by Eq. (199), Eq. (200), and Eq. (201) will be presented. The method avoids any direct matrix inversion by using the matrix inversion lemma [3] to perform the update computations required to account for the presence of the diagonal matrix which is added to the Hessian matrix in the Levenberg-Marquardt algorithm. The development of the algorithm follows.

We start by stating that goal of this method is to compute $\mathbf{P}_{n+1}$ in Eq. (199) without performing any matrix inversion or at least to reduce the inversion problem to a minimum. If we examine Eq. (200) we see that the form of this equation is almost suitable for application of the matrix inversion lemma, as we saw in Chapter 3 in the development of the Recursive Gauss-Newton method. The diagonal $\Delta\mu_{n+1}\mathbf{I}$ matrix in Eq. (200) does not allow for a straight-forward recursive update of $\mathbf{P}_{n+1}$. However, another way to express $\Delta\mu\mathbf{I}$ is:

$$\Delta\mu\mathbf{I} = \begin{bmatrix}1\\0\\\vdots\\0\\0\end{bmatrix}\Delta\mu\begin{bmatrix}1 & 0 & \dots & 0 & 0\end{bmatrix} + \begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}\Delta\mu\begin{bmatrix}0 & 1 & 0 & \dots & 0\end{bmatrix} + \dots + \begin{bmatrix}0\\0\\\vdots\\0\\1\end{bmatrix}\Delta\mu\begin{bmatrix}0 & 0 & \dots & 0 & 1\end{bmatrix} \tag{202}$$

Each term on the right-hand side of Eq. (202) is an outer product. Now consider Eq. (200) as the sum of the $\mathbf{P}_n^{-1}$ matrix, the outer product $\mathbf{j}_{n+1}^T\mathbf{j}_{n+1}$ and $dim(\Theta)$ outer products which represent $\Delta\mu_{n+1}\mathbf{I}$ as seen in Eq. (202). In this form $\mathbf{P}_{n+1}$ can be computed recursively by applying the matrix inversion lemma $dim(\Theta) + 1$ times without any matrix in-

version operations. For the single output case, where M is the number of parameters, the following sequence can be used to recursively compute the LM parameter update :

$$\mathbf{P}_n^{-1} = \mathbf{P}_{n-1}^{-1} + \mathbf{j}_n^T\mathbf{j}_n + \Delta\mu_n\mathbf{I} \tag{203}$$

Then,

$$\mathbf{Q}_1 = \mathbf{P}_n - \mathbf{P}_n\mathbf{j}_{n+1}^T[\mathbf{j}_{n+1}\mathbf{P}_n\mathbf{j}_{n+1}^T + 1]^{-1}\mathbf{j}_{n+1}\mathbf{P}_n \tag{204}$$

$$\downarrow$$

$$\mathbf{Q}_2 = \mathbf{Q}_1 - \mathbf{Q}_1\begin{bmatrix}1\\0\\\vdots\\0\\0\end{bmatrix}\left[\begin{bmatrix}1 & 0 & \dots & 0 & 0\end{bmatrix}\mathbf{Q}_1\begin{bmatrix}1\\0\\\vdots\\0\\0\end{bmatrix} + \frac{1}{\Delta\mu}\right]^{-1}\begin{bmatrix}1 & 0 & \dots & 0 & 0\end{bmatrix}\mathbf{Q}_1$$

$$\downarrow \tag{205}$$

$$\mathbf{Q}_3 = \mathbf{Q}_2 - \mathbf{Q}_2\begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}\left[\begin{bmatrix}0 & 1 & 0 & \dots & 0\end{bmatrix}\mathbf{Q}_2\begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix} + \frac{1}{\Delta\mu}\right]^{-1}\begin{bmatrix}0 & 1 & 0 & \dots & 0\end{bmatrix}\mathbf{Q}_2$$

$$\downarrow$$

$$\vdots$$

$$\downarrow \tag{206}$$

$$\mathbf{P}_{n+1} = \mathbf{Q}_{M+1} - \mathbf{Q}_{M+1}\begin{bmatrix}0\\0\\\vdots\\0\\1\end{bmatrix}\left[\begin{bmatrix}0 & 0 & \dots & 0 & 1\end{bmatrix}\mathbf{Q}_{M+1}\begin{bmatrix}0\\0\\\vdots\\0\\1\end{bmatrix} + \frac{1}{\Delta\mu}\right]^{-1}\begin{bmatrix}0 & 0 & \dots & 0 & 1\end{bmatrix}\mathbf{Q}_{M+1} \tag{207}$$

and,

$$\Delta\Theta_{n+1} = \mathbf{P}_{n+1}[\mathbf{P}_n^{-1}\Delta\Theta_n + \mathbf{j}_{n+1}^T e_{n+1}(\Theta)] \tag{208}$$

and finally,

$$\Theta_{n+1} = \Theta_n - \Delta\Theta_{n+1} \tag{209}$$

For the single output case, Eq. (203) - Eq. (209) can be evaluated without any matrix inversion. This is the stated goal of this method and with respect to matrix inversion the objective of the method has been met.

We must remember that the reason for avoiding a matrix inversion is to reduce the computational burden associated with updating the network parameters at each time step. For this reason the computational requirements for the RLMMIL method must be considered and compared to simply performing an inversion of $\mathbf{P}_{n+1}^{-1}$ to evaluate Eq. (199). Eq. (204) is used to account for the recursive Guass-Newton update at each time step. If this equation is evaluated in the most efficient way, and m is the number of parameters to be adjusted, the following equation for computing the number of floating point operations per sample required applies.

$$\frac{flop}{sample} = 3.5m^2 + 2.5m + 2 \tag{210}$$

This result agrees with Ngia and Sjoberg [22]. Each diagonal element of $\Delta\mu_n\mathbf{I}$ in Eq. (203) is accounted for using Eq. (205) - Eq. (209). If the most efficient method of evaluating each of these terms is used the following equation applies:

$$\frac{flop}{term} = m^2 + 3m + 3 \tag{211}$$

These results are summarized in Table 2.

Method	flop
Recursive Jacobian Update	$3.5m^2 + 2.5m + 2$
Single $\Delta\mu\mathbf{I}$ Term Update	$m^2 + 3m + 3$

Table 2 Computational Requirements for Recursive LM Update (MIL)

In order to compute a full RLMMIL update it is necessary to evaluate Eq. (204) once and an equation of the form of Eq. (205) the same number of times as there are parameters to adjust. The total number of floating point operations required to evaluate $\mathbf{P}_{n+1}$ recursively can be expressed as:

$$\frac{flop}{sample} = 3.5m^2 + 2.5m + 2 + m(m^2 + 3m + 3) = m^3 + 6.5m^2 + 5.5m + 2 \qquad (212)$$

A comparison of the result shown in Eq. (212) with several popular matrix inversion methods [19] which can be used to evaluate the inverse of $\mathbf{P}_{n+1}^{-1}$ is summarized in Table 3. Each of the methods shown in Table 3 requires $O(m^3)$ floating point operations for evaluation. All the methods require more computation than the RLMMIL method with the exception of the Gaussian Elimination Method (GE). Figure 35 shows graphically how for a single recursive LM update, the different inversion methods compare as a function of the number of parameters. The GE method requires 2/3 as many flops as the RLMMIL update. This suggests that there is no numerical advantage in using the RLMMIL method as the GE method requires fewer floating point operations.

Method	flop/sample
Gaussian Elimination	$\frac{2m^3}{3}$
Householder Orthoganalization	$\frac{4m^3}{3}$
Modified Gram-Schmidt	$2m^3$
Bidiagonalization	$\frac{8m^3}{3}$
Singular Value Decomposition	$12m^3$
Matrix Inversion Lemma (Full $\Delta\mu\mathbf{I}$ Update)	m^3

Table 3 Computational Requirements for Full Recursive LM Update

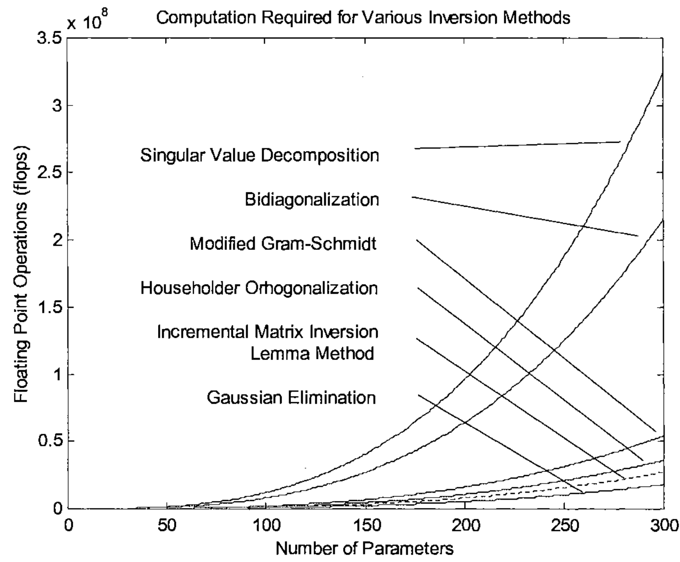


Figure 35 Computational Requirements for Single Recursive LM Update

However, the RLMMIL method incrementally updates the covariance matrix $\mathbf{P}_{n+1}$, opening the possibility of updating $\mathbf{P}_{n+1}$ over several or many time steps. This feature makes it possible to develop an approximate recursive Levenberg-Marquardt method which requires an order of magnitude fewer computations at each time step than a method performing a full $\Delta\mu_{n+1}\mathbf{I}$ update. There are many different ways that the $\Delta\mu_{n+1}\mathbf{I}$ update can be implemented. Several methods for adjusting $\Delta\mu_{n+1}\mathbf{I}$ will be discussed in the next section.

Strategies for adjusting $\Delta\mu$

In the preceding section a method of recursively performing LM parameter updates was developed. It was shown that a RLMMIL update which accounts for all the $\Delta\mu_{n+1}\mathbf{I}$ terms in Eq. (200) is computationally less efficient than using the Gaussian Elimination method to perform the necessary matrix inversion. However, given that there are *m* parameters to adjust, if less than *m* diagonal terms in $\Delta\mu_{n+1}\mathbf{I}$ are accounted for at each time step, far fewer floating point operation are required at each time step. In other words, using the RLMMIL method the Hessian matrix can be updated at every time step using Eq. (204) and one or more $\Delta\mu_{n+1}\mathbf{I}$ terms can be updated at each time step using Eq. (205) - Eq. (207).

For instance, we can see from Table 2 that if only the Hessian update and one diagonal term update is made at each time step the resulting number of floating point operations required is:

$$\frac{flop}{sample} = 3.5m^2 + 2.5m + 2 + m^2 + 3m + 3 = 4.5m^2 + 5.5m + 5 \tag{213}$$

This is approximately an order of magnitude less than the most efficient method of computing the full update. After m time steps have passed, a full accounting of the $\Delta\mu_{n+1}\mathbf{I}$ terms will have been accomplished.

Updates corresponding to a particular location on the diagonal of the $\Delta\mu_{n+1}\mathbf{I}$ matrix can be made sequentially from the first row to the last, or can be randomly chosen, or can be chosen in some other way. Additionally, m updates to the Hessian matrix will have been made. The result of inverting the covariance matrix all at once using a true inverse method will differ from the result obtained using the incremental update.

Due to the adjustment of the parameters at each time step in the incremental method, the rows of the Jacobian will be computed using a different set of parameters at each time step. A constant set of parameters will be used if the full Jacobian matrix is assembled prior to computing the Hessian matrix and then adding it to the $\Delta\mu_{n+1}\mathbf{I}$ matrix before inversion. It should be expected that if the parameter values are reasonably close to convergence, then the elements of the covariance matrix obtained by incremental computation should be fairly close to the values obtained using the true batch LM method. In this situation the parameters will not change dramatically over the m samples used to compute the covariance matrix, so the incremental approximation should be fairly close to the true covariance matrix.

Another approach to making changes to $\Delta\mu_{n+1}\mathbf{I}$ is to adjust only the most significant terms of $\Delta\mu_{n+1}\mathbf{I}$. In this method Eq. (200) must be rewritten as:

$$\mathbf{P}_{n+1}^{-1} = \mathbf{P}_n^{-1} + \mathbf{j}_{n+1}^T\mathbf{j}_{n+1} + \Delta\mathbf{M}_{n+1}\mathbf{I} \tag{214}$$

where,

$$\Delta\mathbf{M}_{n+1}\mathbf{I} = \begin{bmatrix} \Delta\mu_{1,\,n+1} \\ \Delta\mu_{2,\,n+1} \\ \vdots \\ \Delta\mu_{m,\,n+1} \end{bmatrix}\mathbf{I} \tag{215}$$

In Eq. (214) $\Delta\mathbf{M}_{n+1}\mathbf{I}$ is a diagonal matrix containing values which represent the Levenberg-Marquardt diagonal matrix elements, but in this case each element can be different from the other diagonal elements. This deviates from the true Levenberg-Marquardt algorithm, but should still be effective. The challenge with this method is to decide which element to adjust at a particular time step.

Now consider the case where more than one $\Delta\mu_{n+1}\mathbf{I}$ or $\Delta\mathbf{M}_{n+1}\mathbf{I}$ term updates are made during a single time step. In this case, at each time step a Jacobian update is made using Eq. (204) and several updates reflecting a subset of the $\Delta\mu_{n+1}\mathbf{I}$ diagonal terms are made using equations in the form of Eq. (205). It is logical to expect that if several terms are considered at each time step, the approximation to the true covariance matrix will be improved. This is because all the updates made at a single time step are made using a constant set of parameters. This reduces the number of different sets of parameters used in the course of computing a full covariance matrix update. At one extreme, only one $\Delta\mu_{n+1}\mathbf{I}$ term is considered at each time step. At the other extreme, all terms are considered, and the true covariance matrix will be computed. As more terms are considered, a closer approximation to the true covariance matrix will result. This result makes it possible to expand the number of $\Delta\mu_{n+1}\mathbf{I}$ terms considered at each time step to match the available computation-

al capacity for a specific hardware platform or application allowing for maximum efficiency and performance.

Now, again consider the case where more than one $\Delta\mu_{n+1}\mathbf{I}$ or $\Delta\mathbf{M}_{n+1}\mathbf{I}$ term update is made during a single time step. This time, rather than applying the matrix inversion lemma $m + 1$ times to account for each of the $\Delta\mu_{n+1}\mathbf{I}$ terms and the new Jacobian row, the new Jacobian row and/or multiple $\Delta\mu_{n+1}\mathbf{I}$ terms are *simultaneously* accounted for using a "one-step" application of the matrix inversion lemma. The matrix inversion lemma is applied for the case where multiple outer product row updates are made at one time. From Eq. (183) and Eq. (184), for the multiple-term update case, the **A**, **B**, **C** and **D** matrices are defined as:

$$\mathbf{A} = \mathbf{P}_n \tag{216}$$

$$\mathbf{B} = \mathbf{P}_{n+1} \tag{217}$$

$$\mathbf{C} = \begin{bmatrix} \uparrow & 1 & 0 & \cdots \\ | & 0 & 1 & \cdots \\ \mathbf{h}_{n+1}^T & \vdots & \vdots & \ddots \\ | & 0 & 0 & \cdots \\ \downarrow & 0 & 0 & \cdots \end{bmatrix} \tag{218}$$

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & \cdots \\ 0 & \dfrac{1}{\Delta\mu_{n+1}} & 0 & \cdots \\ 0 & 0 & \dfrac{1}{\Delta\mu_{n+1}} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{219}$$

where,

$$\mathbf{B}^{-1} = \mathbf{A}^{-1} + \mathbf{C}^T\mathbf{D}^{-1}\mathbf{C} \tag{220}$$

and

$$\mathbf{B} = \mathbf{A} - \mathbf{A}\mathbf{C}^T(\mathbf{C}\mathbf{A}\mathbf{C}^T + \mathbf{D})^{-1}\mathbf{C}\mathbf{A} \tag{221}$$

If n is the number of outer product terms accounted for by the **C** matrix, then a size n inversion must be performed in order to evaluate Eq. (221). The question is whether or not it is more efficient to compute $\mathbf{P}_{n+1}$ updates simultaneously using the one-step version of the matrix inversion lemma, or to incrementally apply the single outer product version of the matrix inversion lemma multiple times.

Assume that the Gaussian Elimination method will be used to compute $\mathbf{D}$ from $\mathbf{D}^{-1}$ as required for use in Eq. (221). The number of floating point operations required to update one Jacobian matrix row and n $\Delta\mu_{n+1}\mathbf{I}$ terms using the one-step application of the matrix inversion lemma can be expressed as:

$$\frac{flop}{sample} = m^2(2n+3.5) + m(2n^2+5n+2.5) + \frac{2}{3}(n^3+3n^2+3n+1) \tag{222}$$

Eq. (222) indicates that when only a few outer product terms are considered at each time step the impact is minimal. However, as the number of terms considered is increased the impact of the n^3 term has a great impact on the required number of floating point operations.

An interesting comparison between the one-step application of the matrix inversion lemma and the incremental application of the matrix inversion lemma can be made. Table 4 compares the two methods by listing expressions for the required number of floating point

operations for each method as the number of outer product terms (n) considered is held constant at several values.

Number of Diagonal Terms Considered	One-Step Matrix Inversion Lemma (flop/sample)	Incremental Matrix Inversion Lemma (flop/sample)
1	$5.5m^2 + 9.5m + 16/3$	$4.5m^2 + 5.5m + 5$
2	$7.5m^2 + 20.5m + 54/3$	$5.5m^2 + 8.5m + 8$
10	$23.5m^2 + 252.5m + 1331/3$	$13.5m^2 + 32.5m + 32$
50	$103.5m^2 + 5252.5m + 132651/3$	$53.5m^2 + 152.5m + 152$

Table 4 Comparison of One-Step and Incremental Matrix Inversion Lemma

The comparison shown in Table 4 is represented graphically in Figure 36.

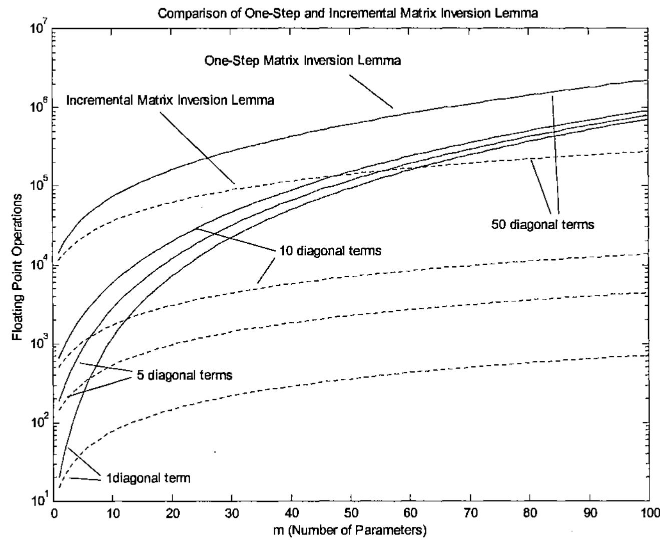


Figure 36 Comparison of One-Step and Incremental Matrix Inversion Lemma

Table 4 and Figure 36 show that no matter how few terms are considered, it is always more efficient to apply the matrix inversion lemma incrementally rather than in a "one-step" fashion. For this reason the one-step approach will not be considered further.

Determining $\Delta\mu_n$

In the preceding section several strategies for adjusting $\Delta\mu_n$ were considered. In order for these algorithms to be useful, an effective method of determining an appropriate $\Delta\mu_n$ adjustment is required. In this section three possible methods for determining $\Delta\mu_n$ will be presented.

In the standard batch LM algorithm an entire set of data is processed by the neural network using the current set of weights. The sum of the squared network output errors (SSE) is computed and compared to the SSE which was computed using the *same* set of data and the previous set of weights. If the SSE goes down, μ_n is decreased by a factor of 10, and the current set of weights is adjusted. If the SSE goes up, μ_n is increased by a factor of 10, and the current set of weights is replaced by the previous set of weights. This process is repeated until a reduction in the SSE is obtained.

In an on-line situation, only a relatively small window of previous input/output data can practically be considered. Additionally, it is impractical to retain a set of input/output data for repeated processing as we can with the batch Levenberg-Marquardt algorithm. Individual data points can realistically only be considered a finite (one or maybe a few) number of times. For these reasons any recursive LM algorithm must incorporate a different set of rules for determining adjustments to μ_n than those used for the batch algorithm.

One problem with real-time adaptive training applications is the fact that system models and noise levels within the processed data change with time. This makes it relatively impossible to make adjustments to μ_n based entirely on changes in absolute error levels. To illustrate this problem consider a simple method of determining $\Delta\mu_n$. Suppose a scheme for adjusting μ_n is devised in which a comparison between the error from one point to the next is used to determine $\Delta\mu_n$. If the error goes up, $\Delta\mu_n$ is negative. If the error goes down, $\Delta\mu_n$ is positive.

Next suppose the network weights have stabilized and the mean squared error has also stabilized. At this point, $\Delta\mu_n$ will tend to be positive because on the average a reduction in the error will not be possible due to noise. In this situation μ_n will effectively continue to grow. Now suppose the process being modeled changes. The neural network errors will become even larger, so the simple algorithm will continue to increase μ_n even though a reduction is probably required so a rapid adaptation to the new model can be achieved. If μ_n is very large, there may not be a fast enough reduction in the MSE to cause a corresponding reduction in μ_n making the adaptive training process ineffective.

One method for determining $\Delta\mu_n$ is to observe the MSE for a window of time prior to adjusting μ_n, then observe the MSE for a non-overlapping window of time after applying $\Delta\mu_n$. A comparison of the average squared error between the two windows of time can then be made. If the MSE from one window of time to the next goes up, $\Delta\mu_n$ will be pos-

itive for the next window of data. If the MSE goes down, $\Delta\mu_n$ will be negative for the next window of data. This method tends to filter out the effect of noise in the training data. This method does not allow for continuous adjustment of μ_n at each time step.

Another approach for determining $\Delta\mu_n$ is to compare the square of the current error measurement to a weighted sum of past squared errors (WSSE). This method allows for adjustment of μ_n at each time step. The idea of this method is to compare the most recent error to an exponentially weighted sum of all past measurements. The weighted sum of all past squared errors can be expressed as:

$$WSSE_{n-1} = e_{n-1}^2\lambda + e_{n-2}^2\lambda^2 + e_{n-3}^2\lambda^3 + \ldots + e_1^2\lambda^n \tag{223}$$

where,

$$0 < \lambda < 1 \tag{224}$$

Computing the WSSE using Eq. (223) gives the highest weighting to the newest errors and causes the oldest errors to be "forgotten".

In order to use Eq. (223) in an algorithm for determining $\Delta\mu_n$ a few adjustments have to be made. Consider the geometric series [19]:

$$1 + q + q^2 + q^3 + \ldots + q^\infty = \frac{1}{1-q} \tag{225}$$

Using Eq. (225), we can approximate Eq. (223) (assuming the squared errors do not change significantly after the algorithm has converged):

$$e_{n-1}^2\lambda + e_{n-2}^2\lambda^2 + e_{n-3}^2\lambda^3 + \ldots + e_1^2\lambda^n \approx \left(\frac{1}{1-\lambda} - 1\right) \tag{226}$$

Eq. (226) indicates that after the neural network weights have converged the most current error multiplied by a scaling factor is expected to equal the infinite power series of past errors if the series contains enough elements. The factor λ effectively determines how many past errors are considered by Eq. (226). The smaller λ is, the fewer the effective number of errors considered.

Eq. (223) can be written for recursive computation in the following way:

$$WSSE_{n-1} = \lambda(e_{n-1}^2 + WSSE_{n-2}) \tag{227}$$

Eq. (226) can be rewritten as:

$$e_n^2\left(\frac{1}{1-\lambda} - 1\right) \approx WSSE_{n-1} \tag{228}$$

Eq. (226) can be used to develop a method for determining $\Delta\mu_n$. If the left-hand side of Eq. (228) is less than the right-hand side for the current sample, then a reduction in μ_n is required. If the left-hand side is higher, then an increase in μ_n is required. The amount of increase or decrease in μ_n can be a fixed, small amount, or can be made proportional to the ratio of the two sides of Eq. (228). Eq. (227) can be used to recursively compute the required *WSSE*. Of course, other variations of this method could also be devised.

RLMMIL Simulation Results

In this section the RLMMIL method will be used to train a neural network in a simulated system identification problem. Figure 37 illustrates the problem.

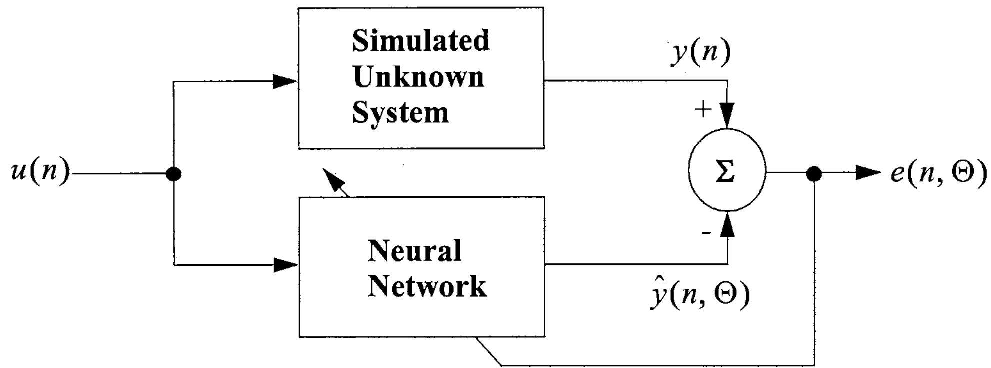


Figure 37 Example RLMMIL System Identification Training Problem

The simulated unknown system is a 3-layer MLP neural network. The input is a tapped-delay line of randomly generated data with a normal distribution. The hidden layers contain log-sigmoid activation functions and the output layer is a single linear neuron. The network had a total of 247 weights. Training data was generated by fixing the network weights at random values and then running the network forward using the random input data to generate the corresponding output data. The neural network to be trained has a similar structure to the network used to generate the training data. Network training was performed using the Steepest Descent, Recursive Gauss-Newton, and RLMMIL methods. The RLMMIL method was applied using three different levels of μ_n updating. In one simulation 20% of the of the $\Delta\mu_n$ term adjustments were made at each time step. In other simulations 50% and 100% of the $\Delta\mu_n$ term adjustments were made at each time step. All of the simulation runs were initiated with the same set of random, small weights. Figure 38 shows the squared-error performance for each method. The plots shown in Figure 38 represents the 100-point moving average squared-error.

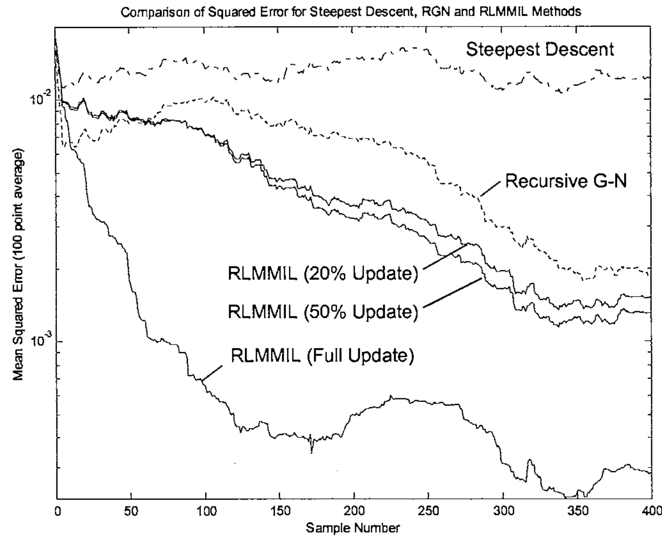


Figure 38 Squared-Error Performance for RLMMIL, RGN and Steepest Descent

Figure 38 shows superior performance for the RLMMIL method as compared to the other methods tested. When the full $\Delta\mu_n$ update was made, the squared-error performance at convergence was very good. When only 50% of the $\Delta\mu_n$ term adjustments were made, the performance decreased considerably. Only a slight reduction in performance was experienced when the number of $\Delta\mu_n$ term updates was reduced from 50% to 20%.

During the simulation runs for the partial $\Delta\mu_n$ term update case, problems were encountered when the size of $\Delta\mu_n$ was too large. If $\Delta\mu_n$ was too large the algorithm would "blow-up". Because only part of the diagonal $\Delta\mu_n$ terms were being adjusted, the magnitude of a portion of the terms on the diagonal might have become disproportionately large or small, causing numerical problems with the recursive inversion equations. The size of $\Delta\mu_n$ had to be decreased more and more as fewer $\Delta\mu_n$ term updates were made at each time step. This effectively causes the method to become more and more like the RGN

method as fewer and fewer $\Delta\mu_n$ updates are considered at each time step. This causes the performance of the algorithm to become the same as RGN when very few terms are considered at each time step. We see evidence of this in Figure 38. As fewer terms are considered at each time step, the performance curves for the RLMMIL method become more and more like that of the RGN method.

The WSSE method described in the previous section was used to determine the $\Delta\mu_n$ adjustments. Figure 39 shows the trajectory for μ_n during the full update RLMMIL simulation.

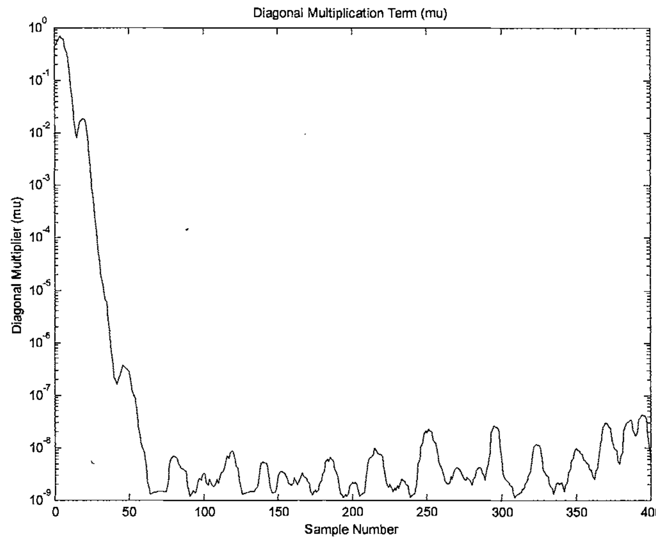


Figure 39 Trajectory of μ_n during training (full $\Delta\mu_n$ case)

Another consideration in evaluating the performance of an optimization algorithm is convergence speed. Figure 40 shows an enlarged view of the performance curves at the beginning of training. The fastest convergence is seen in the Recursive Gauss-Newton curve. The error is reduced the faster, but the RLMMIL error curves quickly drop below the RGN curve to a lower error level. The steepest descent method also has a more rapid

drop in the error curve the RLMMIL method, but shows poor ultimate convergence. More simulation work should be performed to determine if a lower starting value for μ_n, or a more rapid changes in μ_n can improve convergence speed for the RLMMIL method.

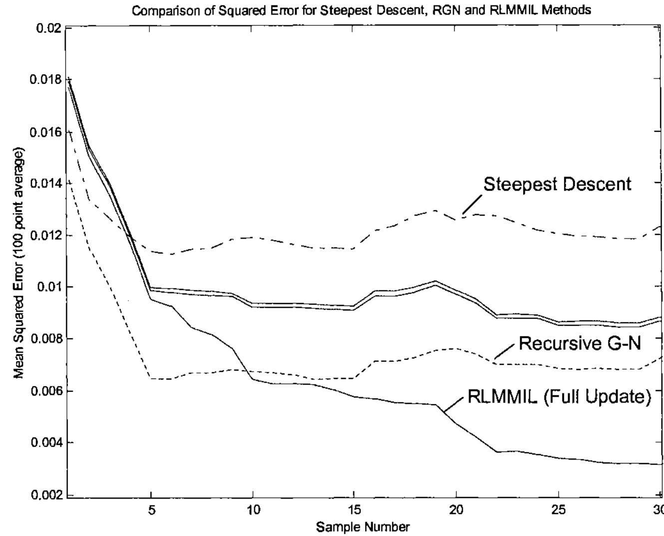


Figure 40 Squared-Error Performance for RLMMIL, RGN and Steepest Descent

Recursive Levenberg-Marquardt with Caley-Hamilton Approximation

In this section a second new method of evaluating the recursive LM algorithm given by Eq. (199), Eq. (200), and Eq. (201) will be presented. The method avoids any direct matrix inversion by using the matrix inversion lemma [3] in conjunction with the Cayley-Hamilton Theorem [19]. The algorithm performs the update computations required to account for the presence of the diagonal matrix which is added to the Hessian matrix in the Levenberg-Marquardt algorithm. The development of the algorithm follows.

As with the RLMMIL method the goal of this method is to compute $\mathbf{P}_{n+1}$ in Eq. (199) without performing any matrix inversion. Repeating Eq. (200) we have:

$$\mathbf{P}_{n+1}^{-1} = \mathbf{P}_n^{-1} + \mathbf{j}_{n+1}^T \mathbf{j}_{n+1} + \Delta\mu_{n+1}\mathbf{I} \tag{229}$$

Eq. (229) can be rewritten as:

$$\mathbf{P}_{n+1}^{-1} = [\mathbf{P}_n^{-1} + \Delta\mu_{n+1}\mathbf{I}] + \mathbf{j}_{n+1}^T \mathbf{j}_{n+1} \tag{230}$$

The matrix inversion lemma can be applied to Eq. (230) using Eq. (183) and Eq. (184). The result for the single output case is:

$$\mathbf{P}_{n+1} = \mathbf{Q}_n - \mathbf{Q}_n \mathbf{j}_{n+1}^T (\mathbf{j}_{n+1} \mathbf{Q}_n \mathbf{j}_{n+1}^T + 1)^{-1} \mathbf{j}_{n+1} \mathbf{Q}_n \tag{231}$$

where,

$$\mathbf{Q}_n = [\mathbf{P}_n^{-1} + \Delta\mu_{n+1}\mathbf{I}]^{-1} \tag{232}$$

The problem arises in computing $\mathbf{Q}_n$ which requires a $dim(\Theta)$ matrix inversion. If $\mathbf{Q}_n$ can be adequately approximated, then is it possible to avoid the matrix inversion. One way to approximate $\mathbf{Q}_n$ is to use the Cayley-Hamiliton Theorem [19]. A statement of the Cayley-Hamilton Theorem follows.

Cayley-Hamilton Theorem

If the matrix $\mathbf{A}$ *is a square matrix having all eigenvalues with magnitudes less than or equal to 1, then*

$$(\mathbf{I} - \mathbf{A})(\mathbf{I} + \mathbf{A}^1 + \mathbf{A}^2 + \mathbf{A}^3 + \ldots) = \mathbf{I} \tag{233}$$

Eq. (233) can be rewritten as:

$$(\mathbf{I} - \mathbf{A}^{-1})(\mathbf{I} + \mathbf{A}^{-1} + \mathbf{A}^{-2} + \mathbf{A}^{-3} + \ldots) = \mathbf{I} \tag{234}$$

and,

$$|\lambda| \le 1 \tag{235}$$

where λ represents the eigenvalues of $\mathbf{A}^{-1}$. Using matrix algebra Eq. (234) can be manipulated to yield:

$$[\mathbf{I}-\mathbf{A}^{-1}]^{-1} = (\mathbf{I}+\mathbf{A}^{-1}+\mathbf{A}^{-2}+\mathbf{A}^{-3}+\ldots) \tag{236}$$

Now, rewriting Eq. (232),

$$\mathbf{Q}_n = [\mathbf{P}_n^{-1}+\Delta\mu_{n+1}\mathbf{I}]^{-1} = \left[\Delta\mu_{n+1}\left[\mathbf{I}+\left(\frac{1}{\Delta\mu_{n+1}}\right)\mathbf{P}_n^{-1}\right]\right]^{-1} \tag{237}$$

and,

$$\mathbf{Q}_n = \left[\Delta\mu_{n+1}\left[\mathbf{I}-\left(-\frac{1}{\Delta\mu_{n+1}}\right)\mathbf{P}_n^{-1}\right]\right]^{-1} \tag{238}$$

If the form of the Cayley-Hamilton theorem given by Eq. (236) is applied to Eq. (238) the result is:

$$[\mathbf{P}_n^{-1}+\Delta\mu_{n+1}\mathbf{I}]^{-1} = \frac{1}{\Delta\mu_{n+1}}\left[\mathbf{I}-\left(\frac{1}{\Delta\mu_{n+1}}\right)\mathbf{P}_n^{-1}+\left(\frac{1}{\Delta\mu_{n+1}}\right)^2\mathbf{P}_n^{-2}-\left(\frac{1}{\Delta\mu_{n+1}}\right)^3\mathbf{P}_n^{-3}+\ldots\right] \tag{239}$$

or, expressed another way,

$$[\mathbf{P}_n^{-1}+\Delta\mu_{n+1}\mathbf{I}]^{-1} = \frac{1}{\Delta\mu_{n+1}}\left[\mathbf{I}+\left[\sum_{k=1}^{\infty}\left[\left(-\frac{1}{\Delta\mu_{n+1}}\right)\mathbf{P}_n^{-1}\right]^k\right]\right] \tag{240}$$

Eq. (239) is only valid if the magnitude of the eigenvalues of $\mathbf{P}_n^{-1}$ are greater than 1. The inverse computed using Eq. (239) can be approximated by considering only the first few terms of the right-hand side. If a finite number of terms are used in the inverse approximation, the RLMCH algorithm can be stated as:

$$\Delta\Theta_{n+1} = \mathbf{P}_{n+1}[\mathbf{P}_n^{-1}\Delta\Theta_n+\mathbf{j}_{n+1}^T e_{n+1}(\Theta)] \tag{241}$$

$$\mathbf{P}_{n+1}^{-1} = \mathbf{P}_n^{-1} + \mathbf{j}_{n+1}^T \mathbf{j}_{n+1} + \Delta\mu_{n+1}\mathbf{I} \tag{242}$$

$$\mathbf{P}_{n+1} = \mathbf{Q}_n - \mathbf{Q}_n \mathbf{j}_{n+1}^T (\mathbf{j}_{n+1} \mathbf{Q}_n \mathbf{j}_{n+1}^T + 1)^{-1} \mathbf{j}_{n+1} \mathbf{Q}_n \tag{243}$$

where,

$$\mathbf{Q}_n = [\mathbf{P}_n^{-1} + \Delta\mu_{n+1}\mathbf{I}]^{-1} \tag{244}$$

and,

$$[\mathbf{P}_n^{-1} + \Delta\mu_{n+1}\mathbf{I}]^{-1} \approx \frac{1}{\Delta\mu_{n+1}}\left[\mathbf{I} + \left[\sum_{k=1}^{L}\left[\left(-\frac{1}{\Delta\mu_{n+1}}\right)\mathbf{P}_n^{-1}\right]^k\right]\right] \tag{245}$$

where L is the number of Cayley-Hamilton terms considered.

Eq. (241) can be used with the standard update equation (Eq. (246) below), to compute the weight updates.

$$\Theta_{n+1} = \Theta_n - \Delta\Theta_{n+1} \tag{246}$$

Any of the methods described previously in this chapter may be used for determining the $\Delta\mu_{n+1}$ adjustment.

RLMCH Method Simulation Results

In evaluating the RLMCH method the first step was to test the accuracy of the Cayley-Hamilton inverse approximation. A randomly generated *dim*(50), symmetric test matrix was generated. To this matrix a diagonal matrix with very small values was added. A standard inversion routine was used to perform the inverse of this matrix. The inverted matrix was then used as the initial $\mathbf{P}_n^{-1}$ matrix in Eq. (245). Eq. (245) was then applied repeatedly, each time increasing μ_{n+1}. At each iteration a sum-of-squared-error computation between the elements of the true inverse and the elements of the Cayley-Hamilton approximation was computed. This test was repeated multiple times using a different number of terms in the Cayley-Hamilton inverse approximation each time. Figure 41 shows the results.

In this example, the C-H inverse approximation is stable for values of μ which are greater than approximately 5.9. Careful investigation has revealed that when μ falls below a value close to 5.9 at least one of the eigenvalues of $\mathbf{P}_n^{-1}$ becomes less that 1 for this example. This violates the conditions under which the Cayley-Hamilton theorem is valid, and the inverse approximation "blows-up". At this point the magnitude of the error increases dramatically as Figure 41 illustrates. When μ is large enough, reasonable approximate inverse results can be obtained. As the number of Cayley-Hamilton terms is increased, the accuracy of the inverse approximation improves. If μ is large, and 5 or more C-H terms are used, the inverse approximation is excellent. Unfortunately, it is desirable to use very small values of μ as the Levenberg-Marquardt algorithm converges.

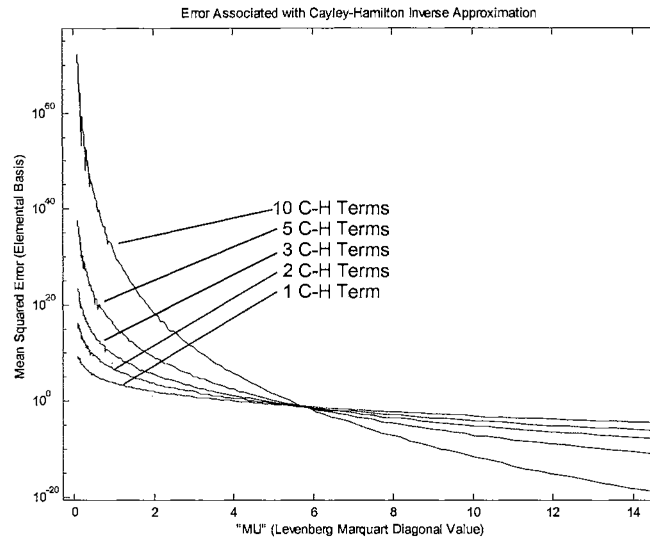


Figure 41 Error Associated with Cayley-Hamilton Inverse Approximation

The next step in testing the RLCH method was to apply the method to an example problem. The same problem which was used to test the RLMMIL method was used to test the RLCH method. This example problem is described in a previous section of this chapter.

Difficulties with this simulation arose immediately. The algorithm was unstable because of the small value of μ which was used to initiate the algorithm. The small value of μ caused the $\mathbf{P}_n^{-1}$ to have at least one eigenvalue with a magnitude less than one, causing the algorithm to diverge. The initial value of μ was increased until the algorithm would start. With such a large value for μ training was very slow. When μ got close to a value of about 100, the algorithm became unstable and diverged. The algorithm was then setup such that μ could never fall below 150. This stabilized the algorithm, but caused training performance to be worse than the results achieved using steepest descent training. After many modifications were made to the LMCH algorithm with unsuccessful results, the de-

termination was made that the Cayley-Hamilton approximate inverse simply can't be used with low enough values of μ for the overall algorithm to be effective. For this reason, no further work on the LMCH method is proposed.

Chapter Summary

The goal of the work presented in this chapter was to develop recursive versions of the popular Levenberg-Marquardt algorithm for on-line training of neural networks. Recent papers by other researchers indicate that there is interest in recursive versions of the LM method. The development of two recursive LM algorithms has been presented in this chapter.

The first algorithm presented is the Recursive Levenberg-Marquardt with Matrix Inversion Lemma (RLMMIL) method. In this method the matrix inversion lemma has been used extensively to avoid having to perform a matrix inversion. It was shown that the RLMMIL method requires an order of magnitude less computation when applied incrementally to account for the LM diagonal matrix terms at each time step. It was proposed that if the correct diagonal term can be chosen for update, the method may be made very efficient and effective. Simulation results showed that the RLMMIL method performed better than the standard Recursive Gauss-Newton but at a higher computational cost. This method shows good promise.

The second method presented in this chapter is the Recursive Levenberg-Marquardt with Cayley-Hamilton (RLMCH) method. This method used both the matrix inversion lemma and the Cayley-Hamilton theorem to compute the incremental Levenberg-Marquardt updates without performing a matrix inversion. Unfortunately this method did not

work well in simulations because of the validity of the Cayley-Hamilton inverse approximation under the matrix conditions required for effective LM training.

Chapter 6

PERFORMANCE COMPARISON BETWEEN ULLS AND COMMON ON-LINE TRAINING METHODS

Introduction

In this chapter a comparison between the ULLS method presented in Chapter 4 and the Recursive Gauss Newton (RGN) and steepest descent methods presented in Chapter 3 will be made. Because the RGN and steepest descent methods are commonly used, they have been chosen for comparison with the ULLS method. The performance comparison is made with respect to squared-error reduction, tracking and computational complexity.

Three different simulation cases will be presented. In the first case all three methods are used to train a three-layer feedforward network using simulated input/output data. In the second case a recurrent network and dynamic backpropagation [23] are used. The third case is a one-step-ahead prediction problem in which a feedforward three-layer network and actual digitized voice data are used.

The ULLS method is the only new training method included in this chapter. This is because this method shows more promise than the other new methods described in Chapter 5 and consequently has been chosen as the best candidate for further research and refinement.

Test Case 1: Non-Recurrent Neural Network Training Using Simulated Data

In this section simulation results for training a typical feedforward neural network using the ULLS and two standard training methods will be compared. There will also be results presented which show the impact on performance when various numbers of past measurements are considered when training using the ULLS method.

The three methods which will be compared in this chapter are gradient descent, Recursive Gauss Newton (RGN) and the underdetermined linearized least squares (ULLS) method. The "moving window" technique of constructing the partial Jacobian matrix was used with the ULLS method throughout the simulations. This method was described previously in Chapter 4 . The performance of the training methods will be evaluated and compared based on speed of convergence, squared-error performance after convergence, and computational burden. The network used in this comparison was a three-layer nonlinear network with 20 inputs, 10 log-sigmoid neurons in the first hidden layer, 3 log-sigmoid neurons in second hidden layer, and a single linear output neuron. Figure 42 shows the network structure used for test case 1.

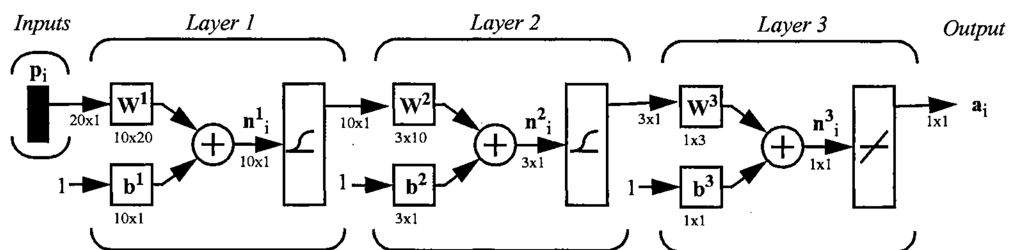


Figure 42 Three-Layer Feedforward Neural Network (Test Case 1)

The network was fully connected and had a total of 247 weights including biases. Training data was obtained by fixing the network weights at random values and running a

forward simulation using a tapped delay structure of random inputs. This data was then used in the training simulations. For each training simulation the network weights were initialized to the same set of small random numbers. The performance of the fixed learning rate gradient descent method was optimized by running simulations with many different learning rates. The comparisons which follow are based on the gradient descent results when using the optimum learning rate. The implementation of the ULLS method used a Jacobian matrix based on a window of 10 past measurements. All simulation results shown here reflect this window size unless otherwise noted. Figure 43 shows the mean squared error for a 200 point window computed at each index in the 3000 training points.

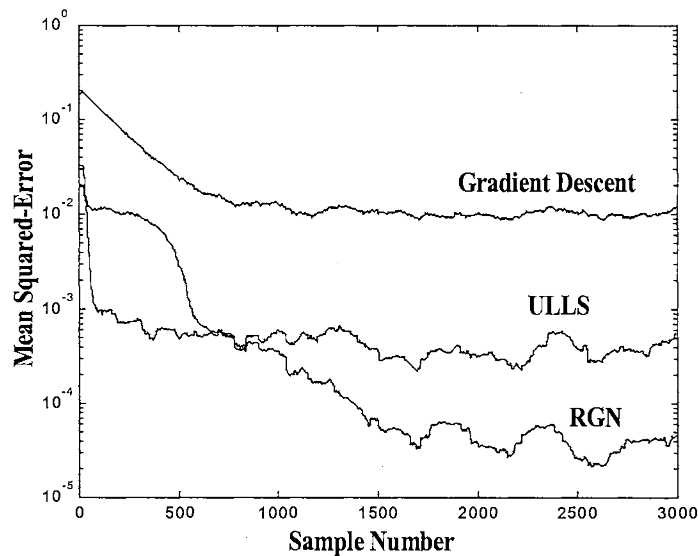


Figure 43 Mean Squared-Error for Three Training Algorithms

We can see from Figure 43 that the gradient descent method converges more slowly and finds a poorer solution than either the RGN method or the ULLS method. It can also be seen that the RGN method converges to a lower mean-squared error than the ULLS method. However, it is very significant that the ULLS method converges much more

quickly than the RGN method. This plot is typical for many different simulation runs using different data sets and initial conditions. We can see from that the ULLS algorithm has essentially converged within approximately 100 points as compared to 500 points for both the Gradient Decent and RGN algorithms. This transient behavior is of great importance in adaptive systems where the model is constantly changing. In this type of application, it is often the case that so long as adequate squared error performance is maintained, convergence speed is more critical than obtaining very small error.

The superior squared-error performance of the RGN and ULLS methods is paid for in computational complexity as indicated in Table 5. The table entries have been computed for each of the methods using all data up to the point of convergence for each individual method. The point of convergence was selected by qualitatively examining the squared error plots.

Training Method	Mean Squared Error (Converged)	Flops/Sample
Gradient Descent	1.00×10^{-2}	1.87×10^{3}
ULLS (10 Jacobian Terms)	1.97×10^{-4}	1.19×10^{5}
RGN	3.77×10^{-5}	2.28×10^{7}

Table 5 Training Method Performance Comparison

These results indicate that with the methods considered, as we would expect, there is improved ultimate squared-error performance as the computational complexity increases. These results also reveal that during the period of adaptation between start and convergence, the ULLS method is the least costly on a squared-error per floating point operation basis.

Figure 44 is plot of mean squared-error vs. floating point operations. As before, the mean squared error for a 200 point window is computed at each index. This plot shows that the gradient descent method exhibits the fastest convergence on a floating point operations basis. However, as shown in Figure 43, the error level after convergence is significantly higher than error levels produced by the RGN and ULLS methods. Notice that the ULLS algorithm can drive the mean squared error much lower than the minimum error achieved by the steepest descent algorithm with less than an order of magnitude increase in required floating point operations.

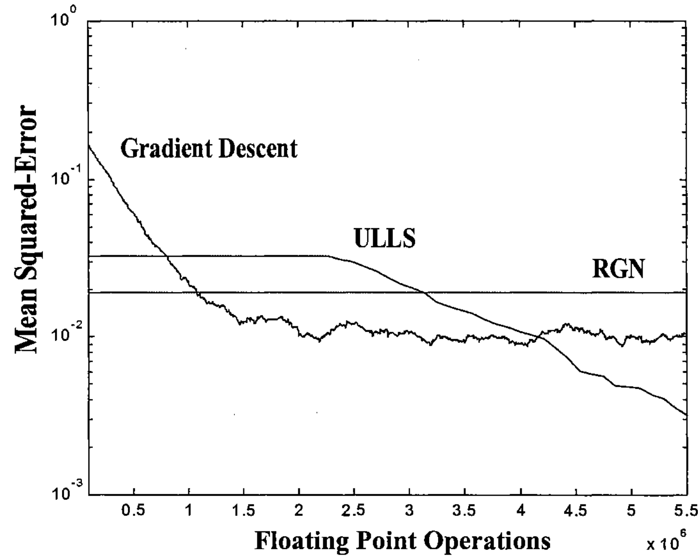


Figure 44 Squared-Error vs. Flops for Three Training Methods

The next set of simulation results reflect the effect of changing the size of the Jacobian window used in the ULLS algorithm. The network structure and data set used for this simulation were the same as for the previous simulations. The size of the Jacobian matrix was varied from run to run. Figure 45 shows the squared-error performance when considering different Jacobian window sizes. It can be seen that, initially, increasing the size of

the Jacobian window has a significant effect on the algorithm performance. As the size of the window is increased, the performance increases significantly in the beginning. In this example there appears to be a point of diminishing returns as the window size is increased to more than about 5 terms. Although not shown, there was little improvement in the squared-error performance when more than about 10 measurements were used in the Jacobian matrix window.

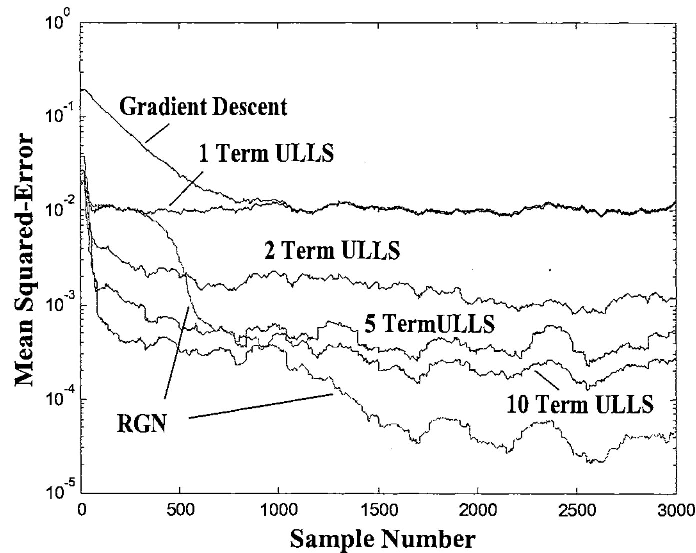


Figure 45 Effect of Jacobian Window Size On ULLS Performance (Non-Recurrent Network)

Table 6 contains a summary of the ULLS algorithm performance and computational requirements for these simulations. Note that the RGN algorithm requires roughly two orders of magnitude more floating point operations than the ULLS algorithm when 10 Jacobian terms are considered.

ULLS Jacobian Terms	Mean Squared Error (Converged)	Flops/Sample
Gradient Descent	1.00×10^{-2}	1.87×10^{3}
1	1.02×10^{-2}	3.08×10^{3}
2	1.03×10^{-3}	7.45×10^{3}
5	3.38×10^{-4}	3.22×10^{4}
10	1.97×10^{-4}	1.19×10^{5}
RGN	3.77×10^{-5}	2.28×10^{7}

Table 6 Jacobian Terms Performance Comparison

Figure 45 shows that the one-term case converges rapidly to the same solution as the manually optimized steepest descent algorithm. This is an interesting and possibly useful result for systems with limited computational power. The single-term implementation provides excellent convergence performance while not even doubling the required computation. If only two terms are used there is a dramatic increase in performance for only a four-fold increase in computational burden. This might also be an attractive improvement for systems with limited computational capacity which currently use gradient descent training.

The next ULLS simulation was conducted to examine the performance of the algorithm when the underlying model parameters are continually changing. In order to accomplish this, test data was generated using the same three-layer neural network structure used previously, but this time the weights in the network were varied sinusoidally between two sets of values. The results are shown in Figure 46. Here we can see that when more Jacobian terms are used, the performance improves. This plot also shows that for this simulation, there is little performance gained by using more than about 5 terms in the Jacobian.

The two-term ULLS MSE plot shows values which are about 25% lower than the steepest descent trajectory, and the five-term ULLS solution is more than 50% lower when the error becomes the largest. The RGN solution shows a very similar result to the best (10-term) RGN solution.

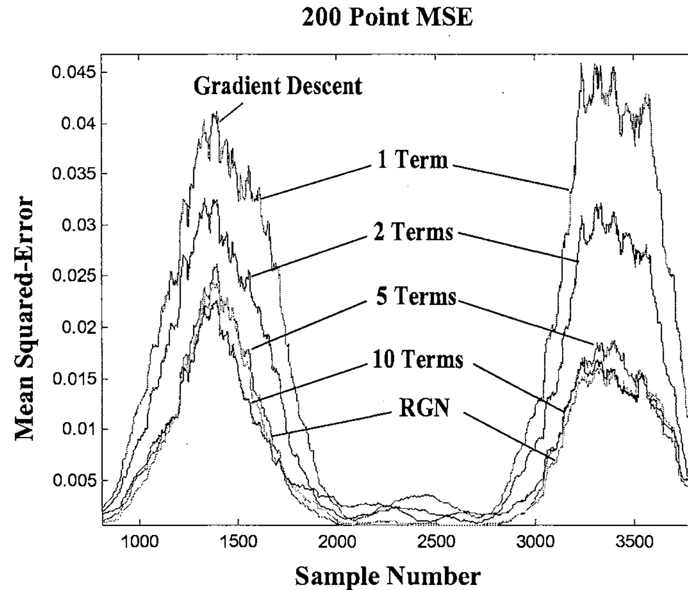


Figure 46 Slow Sinusoidal Parameter Change with ULLS Training

Note that the steepest descent and one-term ULLS MSE trajectories are very similar. This indicates that the algorithms are converging as in the static parameter case (Figure 45.) A faster rate of change in the parameters is necessary to observe the tracking capabilities of the training algorithms.

Another run was made in which the model parameters were changing sinusoidally. In this case the frequency of change in the parameters was increased from every 2000 samples to every 500 samples. It was hoped that the adaptive tracking capabilities of the algorithms would be seen by making this change. The simulation was run in the manner described previously. This time a moving 100 point mean-squared error was computed.

The results are shown in Figure 47. For this simulation we see different error trajectories for the single term ULLS and steepest descent algorithms. Maximum squared-errors are roughly the same for both, but each algorithm outperforms the other at various points in the time-series. When 2 or more Jacobian terms are considered, using the ULLS method, maximum squared-error as well as minimum squared-errors are smaller than those obtained using either steepest descent or the RGN algorithm. The highest squared-errors occur with the RGN algorithm. The change in parameters is apparently too fast for the algorithm to track. This plot illustrates the potential for using the ULLS algorithm in certain cases to obtain lower tracking errors while requiring far less computation than is required for the RGN or other higher-order recursive methods.

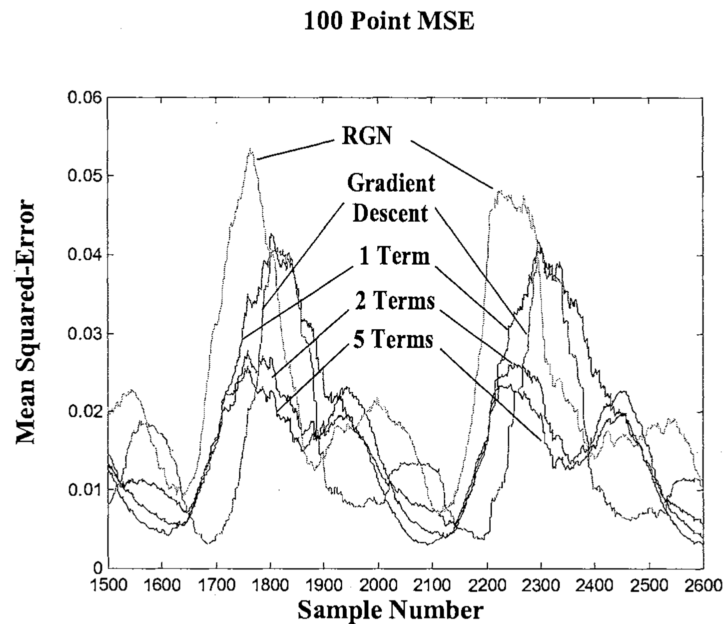


Figure 47 Rapid Sinusoidal Parameter Change with ULLS Training

Test Case 2: Recurrent Neural Network Training Using Simulated Data

In this section simulation results for training a fully recurrent neural network using the ULLS and two standard training methods will be compared. As in the previous section, there will also be results presented which show the impact on performance when various numbers of past measurements are considered when using the ULLS method.

The same three methods (ULLS, RGN and steepest descent) will be compared in this section. The same type of simulations and comparisons made in the previous section will be made in this section. The difference here is that the neural network being trained has a recurrent structure and dynamic backpropagation is used to compute the required derivatives. The "moving window" technique of constructing the partial Jacobian matrix was used with the ULLS method throughout the simulations. The network used in the comparison was a recurrent, three-layer nonlinear network with 10 nonrecurrent inputs, 10 recurrent inputs (20 inputs total), 6 log-sigmoid neurons in the first hidden layer, 3 log-sigmoid neurons in second hidden layer, and a single linear output neuron. The recurrent input vector is formed by placing the output of the network into a tapped-delay-line. Figure 48 shows the network structure used for test case 2.

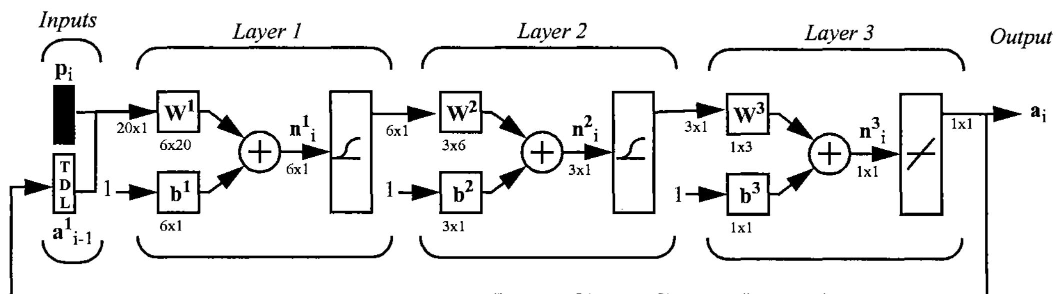


Figure 48 Three-Layer Recurrent Neural Network (Test Case 2)

The network was fully connected between layers and had a total of 151 weights including biases. Training data was obtained in the same manner described previously for test case 1. As before, the implementation of the ULLS method used a Jacobian matrix based on a window of past measurements. A simulation was run to study the performance of the ULLS algorithm when different numbers of Jacobian terms are considered. The results are compared to the performance of the steepest descent and RGN algorithms in Figure 49.

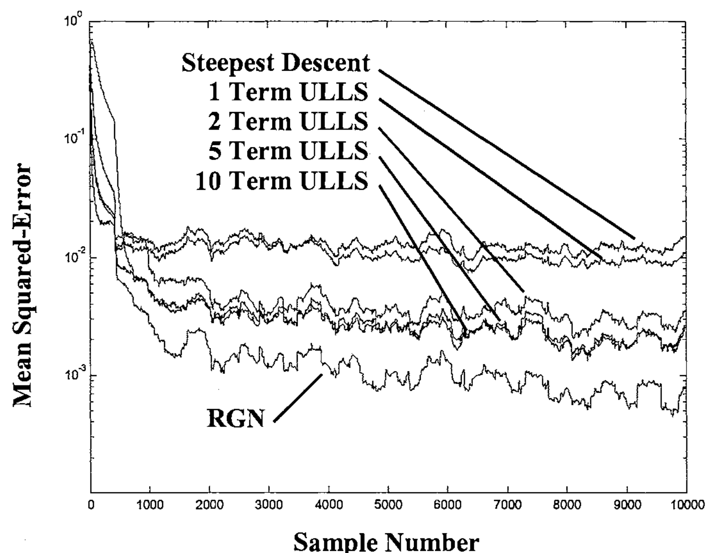


Figure 49 Effect of Jacobian Window Size On ULLS Performance (Recurrent Neural Network)

The results shown in Figure 49 are similar to those obtained for the non-recurrent simulation example (Figure 45). The steepest descent algorithm had the poorest final error performance and the ULLS method performed progressively better as more Jacobian terms were considered. After about 5 Jacobian terms, the benefit of including more terms rapidly diminished. The RGN performance was the best in terms of final convergence error. Both the steepest descent method and the ULLS method converged more quickly than the RGN

method. In terms of computational burden, the steepest descent method had the least and the RGN method had the most. Table 7 provides the computational requirement and mean-squared error performance of each training algorithm.

ULLS Jacobian Terms	Mean Squared Error (Converged)	Flops/Sample
Steepest Descent	1.2752×10^{-2}	4.4402×10^{3}
1	1.0201×10^{-2}	4.9023×10^{3}
2	4.9341×10^{-3}	7.3576×10^{3}
5	4.1395×10^{-3}	2.1230×10^{4}
10	4.2018×10^{-3}	4.4600×10^{4}
RGN	2.5150×10^{-3}	1.3169×10^{7}

Table 7 Jacobian Terms Performance Comparison (Recurrent Network, Fixed Weights)

The next simulation was performed using the same recurrent network structure, but the network weights were varied sinusoidally between two sets of values. The intent was to study the tracking or adaptation capability of the different algorithms. Plots of a moving 200 point mean-squared error for each training method are shown in Figure 50. The highest error was produced by the steepest descent algorithm. The one-term ULLS results are slightly better but similar to those produced by steepest descent training. A dramatic improvement in squared error performance is seen with two-term ULLS training and even more improvement is seen with five-term ULLS training. As we have seen in the previous case, further improvement by using more terms in ULLS training is very small.

The squared-error performance delivered by the RGN method was better than that produced by the one-term ULLS and steepest descent methods, but not as good as the two and five-term ULLS results. It appears that the RGN method is not able to converge quick-

ly enough to adapt to the changing weights. We know from Table 7 and Figure 49 that if the model parameters do not change, that the RGN squared-error performance is superior to that of the ULLS method.

In all the simulations thus far, the RGN method has required approximately two orders of magnitude more floating point operations to implement. This is particularly noteworthy in this example, because the tracking performances of the 2 and 5 term ULLS algorithms are considerably better than that of the RGN algorithm. As with the non-recurrent example presented previously, this example indicates that the ULLS method may offer the potential for improvement where tracking performance is a critical consideration.

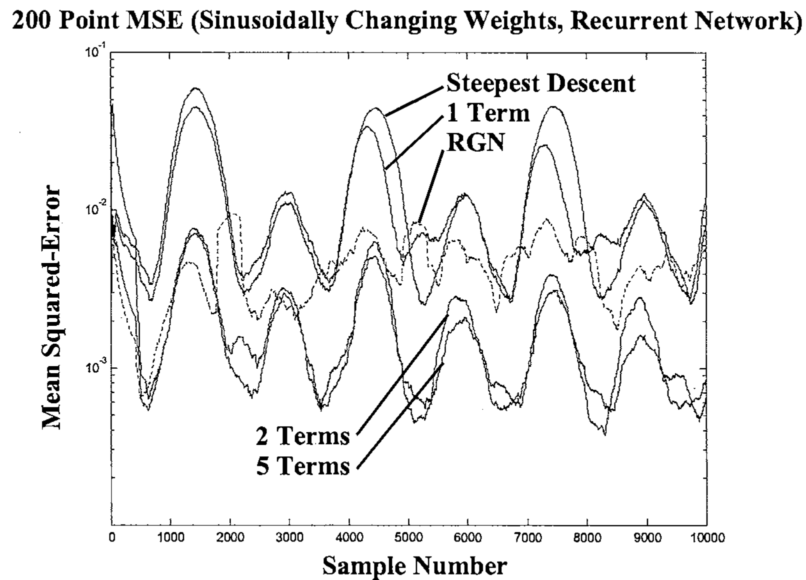


Figure 50 Effect of Jacobian Window Size On ULLS Performance

Test Case 3: Neural Network Training Using Experimental Voice Data

As a final simulation example, a nonrecurrent neural network was used to extract a voice signal which was contaminated with random noise. The idea was to predict the next value in the digitized voice signal. Performance can be measured by computing the squared error between the actual (non contaminated) signal and the prediction from the neural network. The data was sampled at 8000 Hertz. The non-contaminated data plot can be seen in Figure 51.

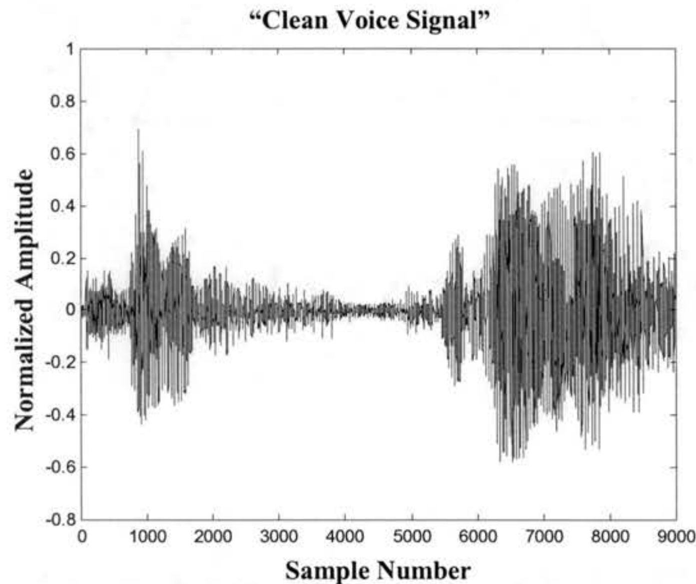


Figure 51 Non-contaminated Voice Signal

Random noise was added to the sample voice signal. The contaminated voice signal can be seen in Figure 52

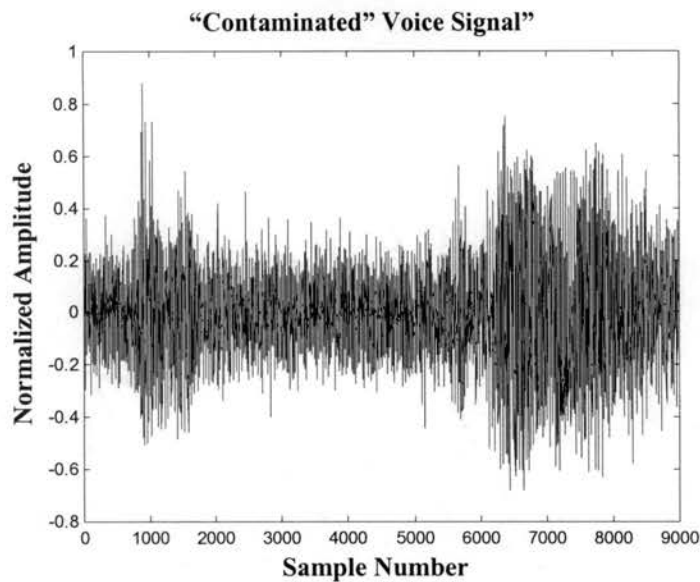


Figure 52 Contaminated Voice Signal

The network used in this example is shown in Figure 42 and has been described in a previous section. The ULLS, RGN and steepest descent training algorithms were used to predict the signal using the data shown in Figure 52. The 200-point squared prediction error between the contaminated and non-contaminated signals is shown for each training method in Figure 53. The results here are similar to those shown for previous simulations. The prediction error was best for the RGN method for most of the time-series. The 2 and 4 term ULLS solutions are very similar to each other, and these are quite close to the RGN solution. The one-term ULLS and steepest descent solution are fairly similar to each other. In this example the difference in performance between the training methods evaluated is much less than it was in the previous examples. This suggests that the ULLS method may not offer large performance increases in every situation. The test results in the previous sections suggest that there are certain types of problems which can benefit considerably from the ULLS

training method.

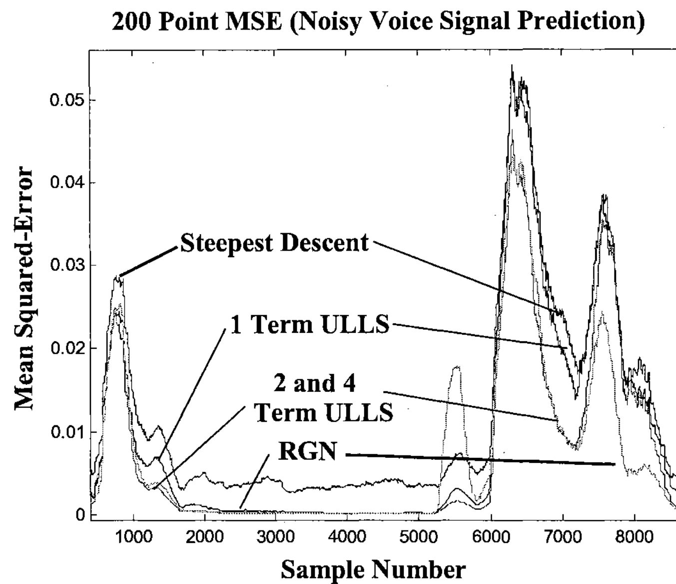


Figure 53 Speech Prediction Performance (ULLS, RGN and Steepest Descent)

Chapter Summary

In this chapter, the Underdetermined Linearized Least Squares (ULLS) training method was compared to the Recursive Gauss-Newton (RGN) and steepest descent training methods. This comparison was made using three different neural network training problems. Simulated data was used for training both non-recurrent and recurrent 3 layer neural networks. In one simulation experimental voice data with added noise was used in a one-step-ahead prediction problem for extracting the "clean" voice signal. The performance comparison has been made with respect to squared-error reduction, tracking and computational complexity.

Simulation results have shown that for training feedforward multi-layer neural networks, the ULLS method presented in Chapter 4 often exhibits faster convergence than ei-

ther steepest descent or Recursive Gauss Newton. The ULLS method also exhibits good final convergence performance, although not as good as the Recursive Gauss Newton (RGN) method. Additionally, the work shows that the ULLS method requires far fewer computations per input sample than the RGN method. If only a few Jacobian terms are considered, the ULLS method requires only a few times as many computations as the steepest descent algorithm. In the simulations shown here, we saw that an order of magnitude reduction in error is possible with the ULLS method over the steepest descent method with only 3 or 4 times as many computations. Although the ULLS method is not capable of driving the squared-error as low as the RGN method when the model parameters are not changing, the ULLS method showed better tracking performance than the RGN method when the model parameters are changing relatively quickly. The ULLS algorithm requires many fewer computations to implement than the RGN method.

The ULLS method dramatically reduces the computational burden and memory requirements associated with standard higher-order on-line training methods. Further reduction of the computational burden associated with the ULLS method may be possible if an efficient method of computing the required inverse which appears in the algorithm is developed.

Chapter 7

APPLICATION OF THE ULLS METHOD FOR DETECTING ROLLER-CONE DRILL BIT FAILURE

Introduction

In this chapter a "real-world" application will be described in which the ULLS algorithm from Chapter 4 is used. Roller-Cone drill bits are a type of rock drilling bit used by the petroleum industry in the construction of oilwells. The objective in this application is to develop a reliable, inexpensive means of early detection and operator warning when there is a roller cone drill bit failure. This system must be technically and economically suitable for use in low-cost rotary land-rig drilling operations as well as high-end offshore drilling. The solution must include the ability to detect impending bit failure prior to catastrophic damage to the bit, but well after the majority of the bit life is expended. In addition to failure detection, the system must be able to alert the operator at the surface once an impending bit failure is detected. The method developed has been proven using experimental test data obtained by running a surface test in which several bits were run to failure.

In this chapter a method in which a predictive neural network is used to detect impending drill bit failure will be described. The method must be suitable for "on-line" implementation in a low-power processor located close to the downhole bit assembly. A description of the problem will be given first, followed by a description of the neural net-

work solution which was developed. Next, the experimental setup used to collect test will be described. Four tests were conducted to obtain experimental data to validate the chosen detection method. In three of these tests bits were run until a failure was obtained. A description of the test set-up and instrumentation for each of the tests is provided. Failure detection results for actual test cases will be presented next. A brief description of the method used to signal the surface operator will appear next, followed by conclusions.

Problem Description

When drilling a well it is desirable to drill as long as possible without wearing the bit to the point of catastrophic bit failure. Optimum bit use occurs when a bit is worn sufficiently that the useful life of the bit has been expended, but that wear is not so extensive that there is a high likelihood of mechanical failure which might result in leaving a portion of the bit in the well. Poor drilling performance, increased BHA (Bottom Hole Assembly) wear, and more frequent debris recovery operations all result from continued drilling with bits which are in the process of mechanical failure. A system capable of detecting the early stages of bit failure, with the additional capability of warning the operator at the surface would be of great value solving the problem of drilling to the point of catastrophic bit failure.

The first part of the problem is to detect a bit failure in progress. The solution that was chosen utilizes a downhole sensor sub that contains accelerometers that monitor the acoustic signals produced by the bit. The signals produced by the bit are processed by a microprocessor contained in the sensor sub. An algorithm named the Adaptive Neural Network Prediction Analysis (ANNPA) method has been developed which is capable of pro-

cessing the bit signals and detecting a bit failure in progress. The ULLS algorithm is a key component of the ANNPA failure detection algorithm.

Several methods of failure detection were considered. It appears that some work has been done on placing sensors directly in the drill bit assembly to monitor the bit condition. There is some merit in placing sensors in the bit assembly, but this methodology also has some distinct disadvantages. The main disadvantage is the necessity of redesigning every bit which will use the method. In addition to being costly, each new bit design will have to accommodate the embedded sensors which might compromise the overall mechanical design. A second disadvantage arises from the fact that sensor connections and/or data transmission must be made across the threaded connection on the bit to a data processing or telemetry unit. This is difficult in practice.

The ANNPA failure detection method can be considered an "indirect" method of detection in which the sensors used to measure signals produced by the bit are located directly above the drill bit in a special sensor/telemetry sub and not within the bit itself. Additionally, the measurements that are being made are not direct measurements of bearing parameters (i.e. wear, position, journal temperature etc.), but of a symptom of bit failure such as vibration. This type of arrangement has some very desirable features. The most significant advantage of this method over other methods is the characteristic that this method may be used with *any* bit without modifying the bit design in any way. This effectively separates the bit design from the detection/warning system so the most desirable bit design can be achieved without concern for the accommodation of embedded sensors. Figure 54 illustrates the physical arrangement of the sensors in relation to the bit.

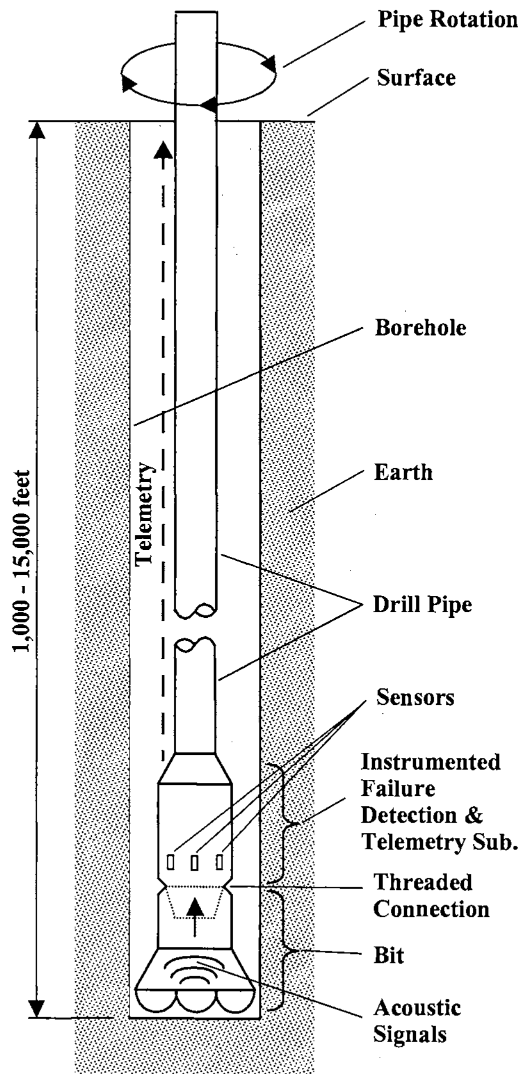


Figure 54 Roller-Cone Drill Bit Bearing Failure Detection

In the ANNPA method, an adaptive neural network is used to process sensor signals as part of an overall scheme to detect drill bit failure. Figure 55 shows a schematic of the failure detection system. Sensor signals are received by the neural network, which uses past signal measurements to predict the next sensor value.

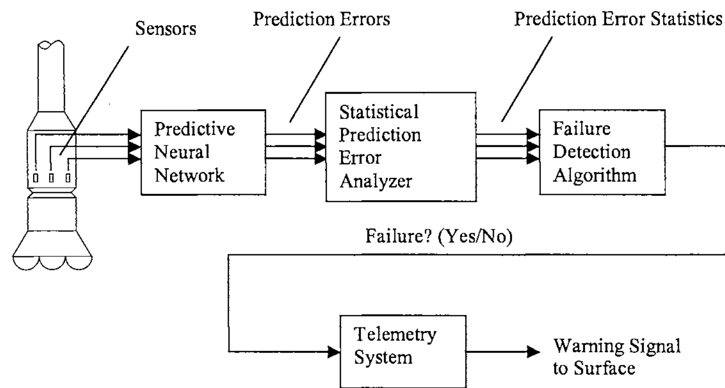


Figure 55 Schematic of ANNPA Drill Bit Failure Detection Scheme

The neural network attempts to predict sensor readings one step ahead in time using older sensor readings. Figure 56 shows the sensor data prediction scheme using a neural network.

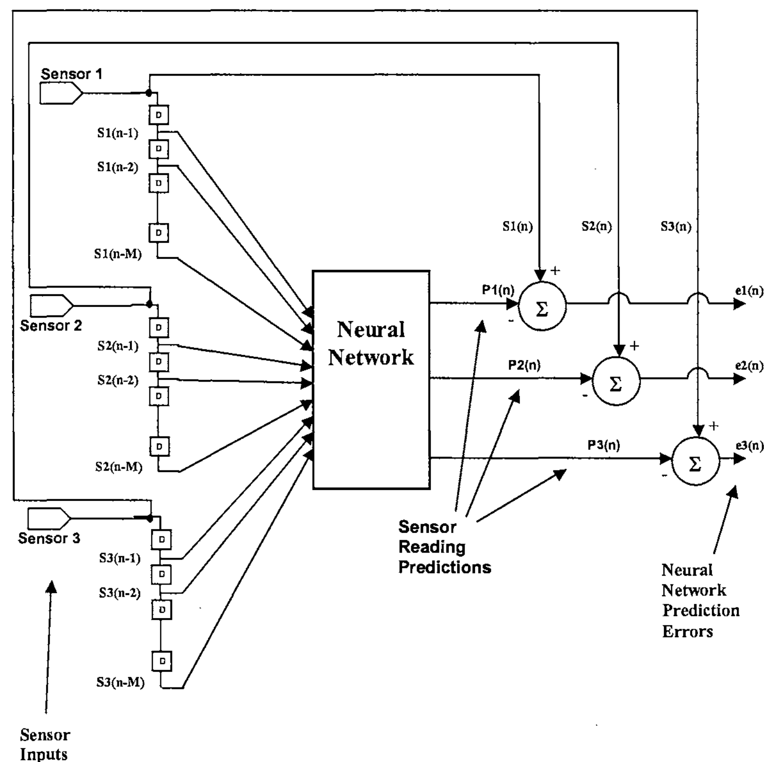


Figure 56 Adaptive Neural Network Predictor (ANNPA Method)

The past sensor values are stored in a tapped-delay-line memory structure. These values are then used as inputs to the neural network. The neural network then predicts the next value expected from each of the sensors. The value predicted for each of the sensors is then subtracted from the actual sensor readings to compute a prediction error. If the neural network prediction is good, the computed prediction error will be small. If the prediction is poor, the prediction error will be high. The square of the prediction error is computed and analyzed. If the signal being predicted is fairly repetitive (periodic) it is possible to successfully predict future signal values. If there is a large random component in the signal being predicted, or if the nature of the signal changes rapidly, it is difficult to successfully predict future signal values. The ANNPA method exploits this characteristic to detect bit failures.

Under normal drilling conditions with a bit in good condition, the vibration in the bit is fairly periodic with a significant random component added in. If a neural network prediction is performed on a time-series of vibration measurements taken near the bit, there will be a level of prediction error which does not change rapidly over a short period of time. This is because the neural network will be capable of predicting much of the periodic vibration associated with the bit. However, random vibrations due to the drilling environment such as rock type, fluid noise, etc. will not be predictable and will result in prediction errors. Test data has shown that when a bearing in a cone starts to fail, it will generally emit bursts of high-frequency vibration or will cause the cone to lockup. Either of these occurrences will cause an abrupt and unpredictable change in the pattern of vibrations produced by the bit. If the prediction error of a neural network that is being used to predict bit vibration is

monitored, momentary increases ("spikes") in the prediction error will be observed. These observations can be used to detect roller cone bit failure. Figure 57 is an illustration of the prediction error for normal running conditions and spikes in the prediction error related to failures.

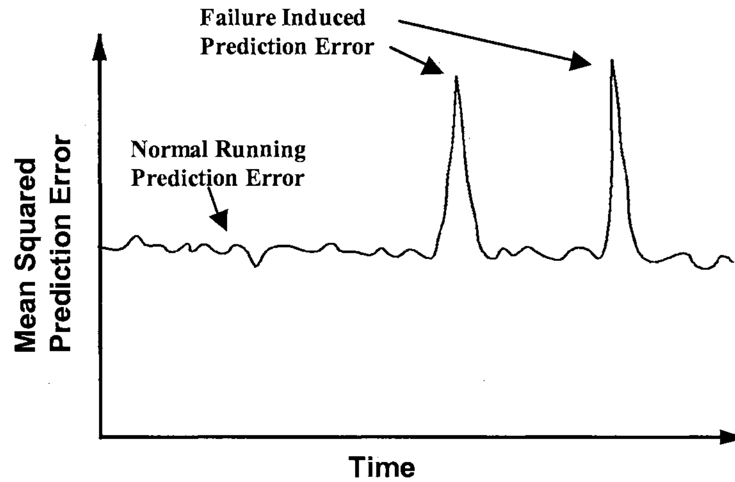


Figure 57 Failure Indications (ANNPA) Method

One way to determine if a failure is in progress is to look for spikes in the prediction error which exceed a threshold value with an average frequency of occurrence that also exceeds a threshold frequency value. In other words if a high enough spike in the prediction error occurs often enough this means there is a failure in progress. Another way to detect failure is to monitor the standard deviation of the prediction error. If the standard deviation gets large enough, a failure is indicated. In addition to monitoring a threshold value for the prediction error it might be useful to monitor the *change* in prediction error. These methods are examples of potential ways to analyze the neural network prediction error to detect bit failure.

All signal processing must be performed at the drill bit in real-time. Limited processing capability and low available power make it important to use an efficient training method such as the ULLS algorithm.

ANNPA Method Experimental Verification

To verify the validity of the ANNPA method, a neural network was trained using the ULLS method to predict actual acoustic signals produced by a roller cone bit before and during failure. Experimental data was collected from a laboratory test of an actual drill bit in operation. In this section the performance results of the ANNPA method when applied to experimental data will be presented. Experimental data was collected while using an actual roller cone bit to drill into a cast iron target. Sensors were mounted to a sub directly above the bit and a data acquisition system was used to record the sensor readings. Accelerometers were attached to the sub directly above the bit. A tri-axial accelerometer was used. The bit was held stationary and loaded vertically into the target while the target was turned on a rotary table. Figure 58 shows a picture of the experimental test setup.

The sampling rate for most of the data recorded was 5000 hertz. Test data was recorded at sample rates of 5000, 10,000, 20,000 and 50,000 hertz. A frequency analysis showed that a very high percentage of the total signal power was below 2000 hertz. For this reason and to reduce unnecessary data storage, a sample rate of 5000 hertz was used for the series of tests.

An IADC class 117W 12-1/4" XP-7 bit was used for all tests. The test procedure consisted of flushing the number 3 bearing with solvent to remove most of the grease and then running the test bit with a rotational speed of 60 rpm and a constant load of 38,000

pounds. Cooling fluid was pumped over the bit throughout the test. Under these drilling conditions the contamination level in the number three bearing was increased in steps.

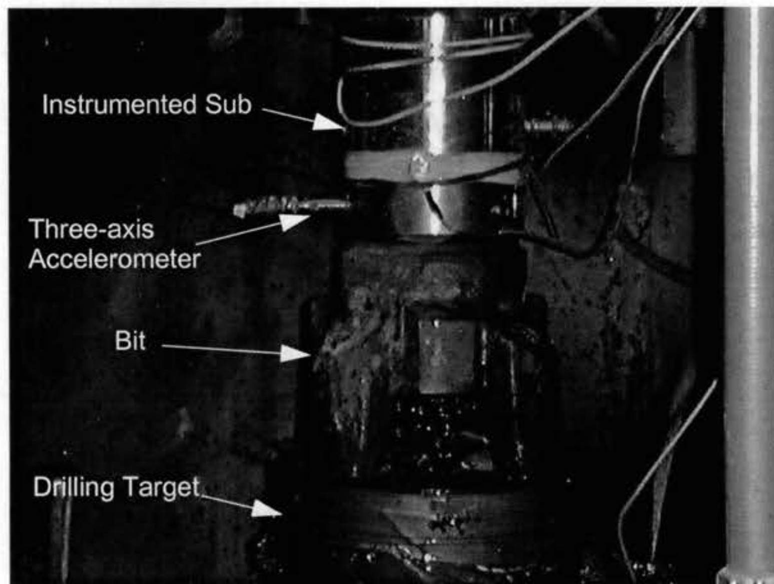


Figure 58 Experimental Test Setup

This process continued until the number 3 bearing was very hot, and was beginning to lock up. Baseline data with the bit in good condition and the bearing at a low temperature was taken before any contamination was introduced to the bit. A section of the baseline (no bearing damage) data is shown in Figure 59.

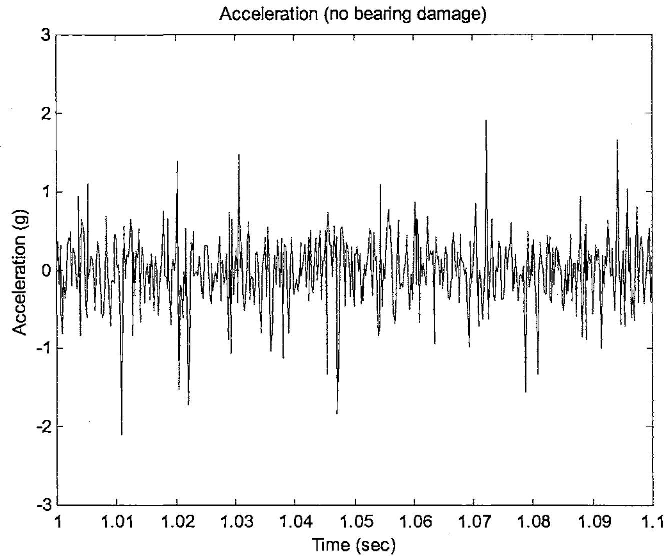


Figure 59 Acceleration (no bearing damage)

A three-layer neural network containing log-sigmoid activation functions and a 20 tap input structure was trained to predict the acceleration signal using the ULLS method. There were 5 neurons in the first layer, 3 neurons in the second layer, and one linear output. Three different neural networks were trained, each using a different axis of acceleration as a target. There results for each neural network were very similar. A sample of the resulting prediction error is shown in Figure 60. As Figure 60 reveals, the prediction error was fairly consistent and small when there was no bearing damage.

Testing continued for several hours. Twice during the test a drilling mud mixture consisting of 1.4 liters of water, 100 grams of bentonite and 1.1 grams of sodium hydroxide was pumped into the number 3 bearing area. After the addition of the mud and after extended drilling some bearing failure, occasional "spikes" in the accelerometer data indicat-

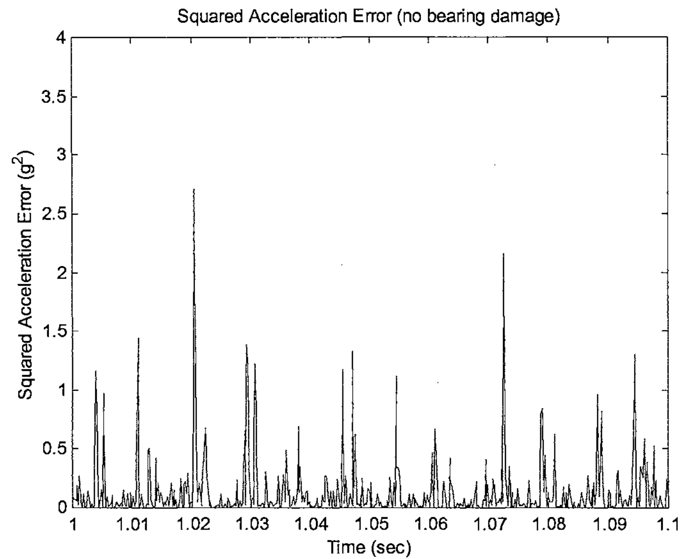


Figure 60 Squared Prediction Error (no bearing damage)

ed early bearing failure. Figure 61 and Figure 62 show accelerometer data and the corresponding neural network prediction error. In Figure 62 the "spikes" in the prediction error indicate early bearing failure. This was verified by close inspection of the bit.

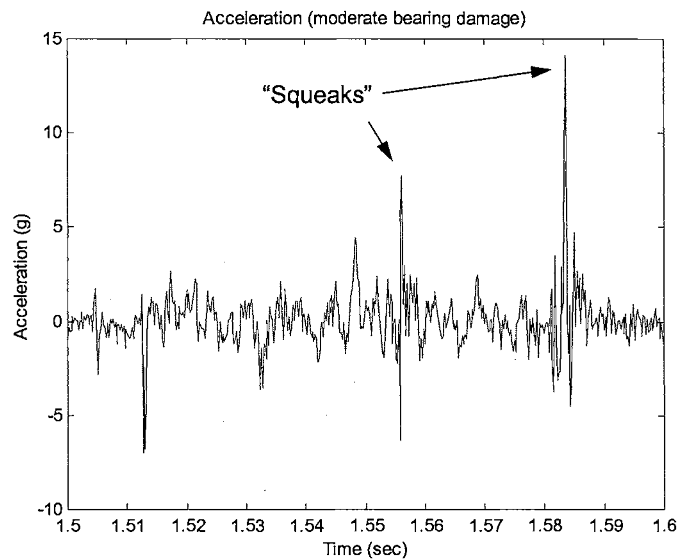


Figure 61 Acceleration (moderate bearing damage)

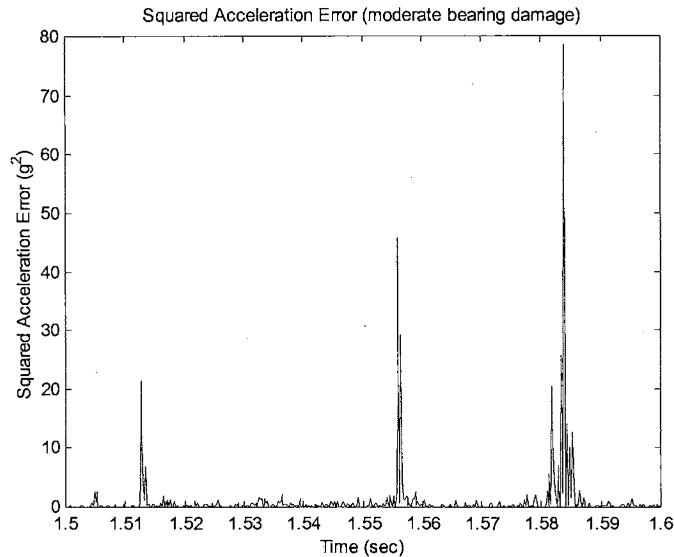


Figure 62 Squared Prediction Error (moderate bearing damage)

In the last phase of the test, drilling was halted and a solution of 1.4 liters of water, 100 grams of bentonite, 1.1 grams of sodium hydroxide, and about a gram of sand was pumped into the number 3 bearing area. Drilling resumed, and the bearing quickly began to show signs of increasing failure. The number 3 bearing began to produce steam as it heated up. Figure 63 and Figure 64 show the accelerometer data and prediction results for the data recorded under these conditions.

The last test data was recorded after significant bearing wear. This data was recorded just prior to bearing lockup. The "squeaking" in the bearing is obvious in Figure 63. Numerous failure indications can be seen in Figure 64. It must be noted that the "slop" (increase in bearing clearance) in the number 3 bearing is still very small. This means that a very definite failure detection was indicated long before catastrophic bearing separation.

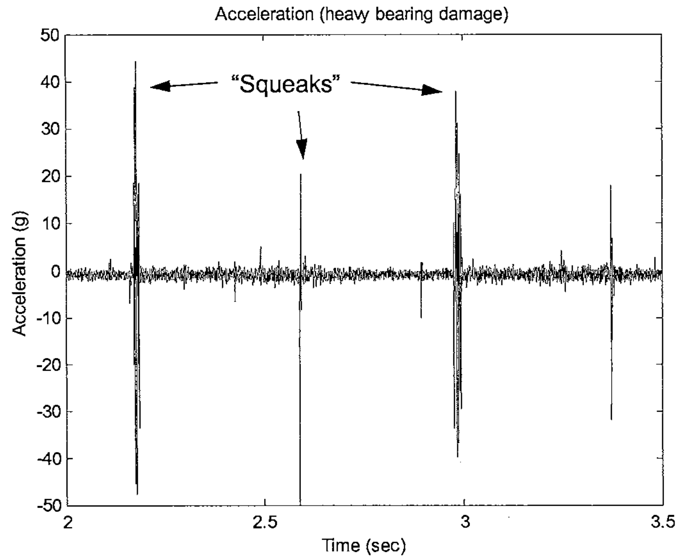


Figure 63 Acceleration (heavy bearing damage)

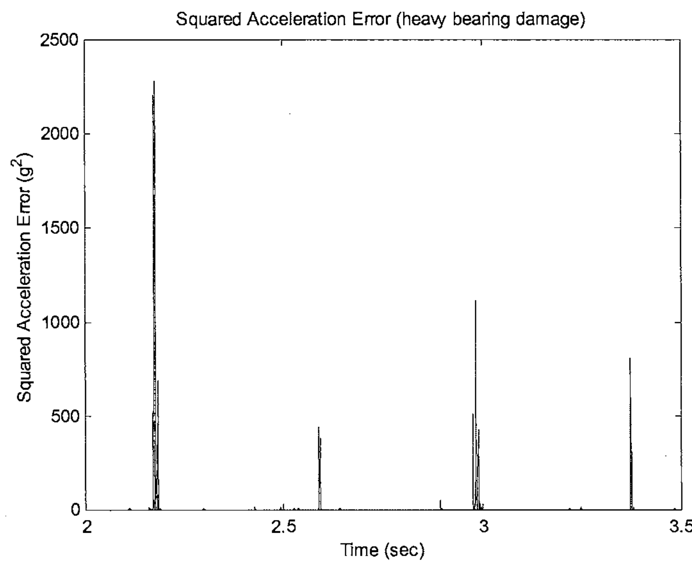


Figure 64 Squared Prediction Error (heavy bearing damage)

Downhole Tool and Warning System Description

In the first section the method of detecting bearing failure was described. In this section a method and apparatus for signaling the operator at the surface will be described. The general idea is the following. Under normal rotary drilling operations surface pump

pressure is applied to the drill string which creates a high-pressure jet via nozzles in the drill bit. This is also true when drilling is performed using a mud motor. A large pressure drop is present across the nozzles in the bit. For example, a pump pressure of 2500 psi might be applied to the drill string at the surface. This applied pressure will be seen at the bit, minus fluid friction and other pressure losses. So the flowing pressure drop across the bit might be around 1200 psi. If a non-restrictive port is opened above the bit, the flowing pressure within the entire system will be reduced. In other words, if a large port is opened above the bit, the 2500 psi applied at the surface will drop to say 1800 psi. This pressure drop can be used as a signal to the operator that the port has opened indicating a particular condition downhole such as a bearing failure. Figure 65 illustrates the process.

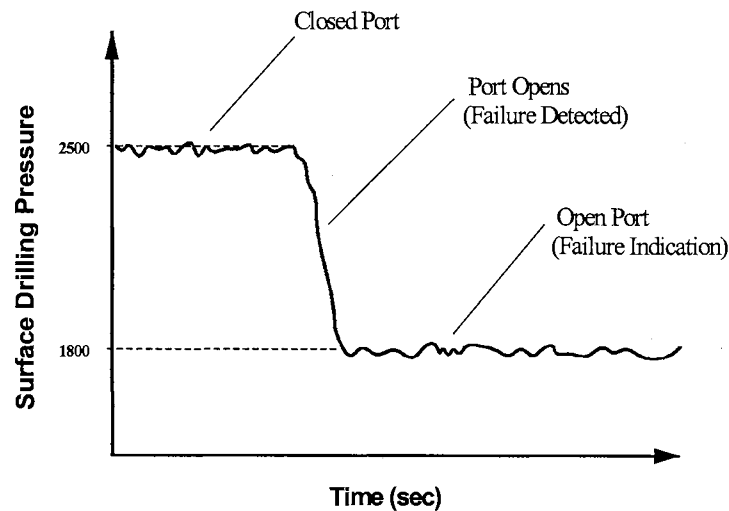


Figure 65 Open Port Failure Indication

The basic detection/warning system operation follows a sequence. First the sensor data is monitored while the drilling operation proceeds. The detection method previously described is used to detect a failure in progress. If a failure is detected a port is opened which causes a drop in the surface pump pressure. This drop in pressure can easily be seen

by the surface operator, serving as a warning that a failure is in progress in the bit. A schematic of the downhole tool apparatus is shown in Figure 66.

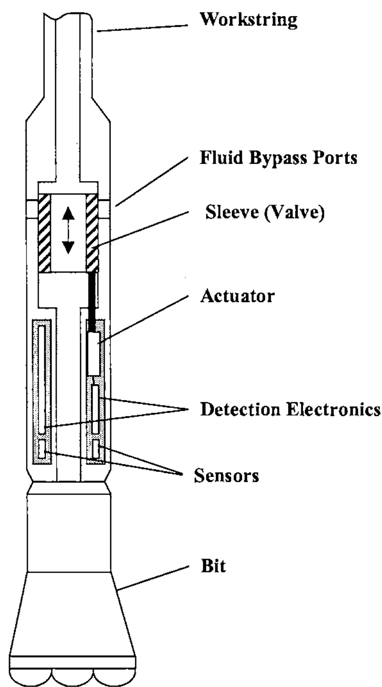


Figure 66 Downhole Tool Schematic.

In this device, a sleeve valve can be opened and closed repeatedly to cause corresponding low and high pressure pumping pressure levels at the surface. A microprocessor or digital signal processor is used to implement the detection algorithm and monitor the sensors. Additionally the processor will control the actuator, which open and closes the sleeve valve. It may be desirable in some cases to close the bypass valve after a certain delay, so normal drilling can proceed if desired. Figure 67 shows the surface pressure sequence associated with this type of operation.

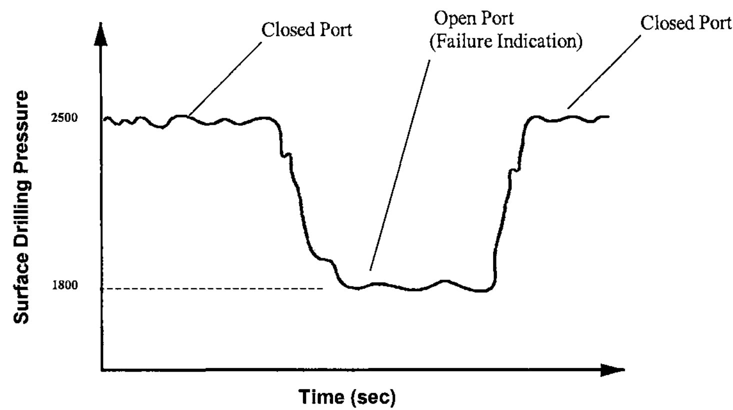


Figure 67 Open-Close Signaling Operation

Chapter Summary

During drilling operations it is important to avoid wearing a bit until catastrophic failure occurs, leaving a portion of the bit in the well. In this chapter a bearing failure detection system which utilizes the ULLS algorithm to train a multi-layer neural network has been described. The ULLS algorithm is suitable for this application because of the limited computational capability at the bottom of the well where sensing equipment and the failure algorithm must operate in "real-time". This method has been tested with positive results using experimental data recorded during laboratory tests in which a drill bit was tested to the point of failure.

This chapter also describes a relatively simple method of providing a surface indication to the operator that a failure has been detected. In this method surface pump pressure across the bit is altered to send information from downhole to the operator.

Chapter 8

CONCLUSIONS AND RECOMMENDATIONS

Conclusions

In early chapters, background information on standard optimization methods was presented. These methods have been used in one form or another to train neural networks. In Chapter 2, batch optimization methods were discussed. These batch training methods generally provide the basis for the development of most of the on-line or incremental optimization methods presented in Chapter 3. In most cases, simple examples were used to demonstrate each algorithm. This background material is necessary, as it has been the starting point for new algorithms which were developed during this research.

The new results presented in this work can be separated into two parts. The first part is the development of the new Underdetermined Linearized Least Squares (ULLS) algorithm for training neural networks and a recursive form of this algorithm the Recursive Underdetermined Linearized Least Squares (RULLS) algorithm. In Chapter 7, the ULLS algorithm was applied to a real-world problem involving the detection of bearing failures in oilwell drill bits.

The second new result includes one approximate and one more exact method for implementing the Levenberg-Marquardt algorithm recursively.

In a general sense, the objective of this work has been to develop new on-line, recursive algorithms for training neural networks for use in adaptive filtering and controls applications. Specifically, one goal of the research has been to develop a recursive version of the well-known Levenberg-Marquardt algorithm. The motivation for this goal is the excellent performance given by the batch Levenberg-Marquardt algorithm. It is an attractive proposition for adaptive filtering and controls applications to have such an algorithm available in an efficient, recursive form. The work on a recursive Levenberg-Marquardt algorithm led to a second primary research goal. This has been to develop a new training algorithm based on the underdetermined solution to the linearized least-squares case with improved performance and modest computational requirements. The need for an algorithm of this type is particularly noticeable in neural network filtering and controls applications where tapped-delay-lines and fully connected networks make it necessary to use many parameters which must be adjusted at each time step. A method which allows for the adjustment of many parameters while only using a relatively small number of samples might prove to be efficient and useful in these applications.

The main contribution of this work is the ULLS (underdetermined linearized least squares) algorithm and a recursive form of the RULLS algorithm. The application of the underdetermined solution to the training of neural networks appears to be a new method not previously noted in the literature. The basis for this algorithm is the standard linear least squares solution. The least-squares solution to the estimation of parameters for the underdetermined (fewer parameters than samples) linear case was presented. It was then shown that by linearizing a nonlinear function such as a neural network, then applying the solution

to the linear underdetermined case, that parameter estimates for the nonlinear function could be made with fewer samples than parameters. This concept was developed into a method of training neural networks using a “window” of past measurements to form a “moving” Jacobian matrix for use in the ULLS algorithm. This window of Jacobian terms provides a mechanism for “forgetting” past measurements, making the algorithm adaptive.

The algorithm is particularly useful for on-line training of neural networks. In filtering and controls applications it is typical to use a tapped-delay-line input structure. When a linear function (i.e. a finite impulse response filter) is used, the number of parameters which must be adjusted is equal to the number of inputs. In typical applications the number of “taps” or inputs can range from tens to hundreds in number. Now suppose the linear function is replaced by a fully-connected nonlinear neural network. For the same number of taps, the number of parameters increases at least 100% for each neuron in the first layer. Additional parameters which are associated with the other layers of the network are also added. If standard higher-order training algorithms such as Recursive Gauss-Newton or Extended Kalman Filter are used, the size of the matrices which must be manipulated equals the number of parameters. When tapped-delay-lines are used with fully-connected neural networks the number of parameters can be prohibitively large for use with standard higher-order training algorithms. Problems arise with respect to both computational burden and memory requirements. The ULLS algorithm allows many parameters to be adjusted using only a few observations with reduced computational burden.

In test cases, the performance of the ULLS algorithm has been shown to be superior to the steepest descent method while only requiring modest increases in computational complexity and memory requirements.

In order to make the ULLS algorithm even more efficient, a fully recursive form of the algorithm was developed using the block matrix inversion algorithm. The RULLS algorithm does not require the inversion of a matrix. However, numerical simulations of the RULLS algorithm revealed a susceptibility to numerical instability due to the finite precision of a digital computer. This problem was examined, but unfortunately a simple solution was not discovered.

The other contribution of this work was the development of a recursive form of the Levenberg-Marquardt batch optimization algorithm. The goal of this work was to develop a recursive LM algorithm which requires no direct matrix inversion thereby making the algorithm as computationally efficient as possible. Two different approaches were taken in this effort.

The first method, called the RLMMIL (Recursive Levenberg-Marquardt Using Matrix Inversion Lemma) method, avoids any direct matrix inversion by using the matrix inversion lemma to perform the update computations required to account for the presence of the diagonal matrix which is added to the Hessian matrix in the Levenberg-Marquardt algorithm. In this method the matrix inversion lemma is applied once for every diagonal term added to the Hessian matrix. This means that if there are N parameters, the matrix inversion lemma must be applied N times. It has been shown in simulations that the solution produced using the RLMMIL method is the same as the solution obtained by perform-

ing the inversion directly. It was also shown that the computational requirements for a full update using the RLMMIL method are lower than all standard inversion techniques except Gaussian Elimination. However, if a partial update is made using the RLMMIL method in which a limited number of the diagonal LM terms are computed, the computational requirement can be reduced by approximately an order of magnitude. The key here is deciding which term(s) should be updated. This refinement to the method was not completed, but this method shows much promise.

A second approach was taken in developing a recursive Levenberg-Marquardt algorithm. The method avoids any direct matrix inversion by using the matrix inversion lemma in conjunction with the Cayley-Hamilton Theorem . The idea here was to approximate the required inverse by using a form of the Cayley-Hamilton theorem which fits the form of having a diagonal matrix added to the Hessian matrix as we see in the Levenberg-Marquardt algorithm. Unfortunately, the Cayley-Hamilton theorem only holds for certain matrix conditions which are usually violated as the algorithm converges.

Recommendations

There are a number of areas in which the present work could be improved. The new training algorithms developed in the course of this research all offer substantial opportunity for further refinement and improvement.

Recursive Form of the ULLS Algorithm

The Underdetermined Linearized Least Squares (ULLS) algorithm was shown to be an effective algorithm for training nonlinear networks. A recursive form of this algorithm, the RULLS algorithm was developed with the goal of reducing the computational complex-

ity of the ULLS algorithm. Unfortunately, the RULLS algorithm as it stands, suffers from numerical instability when implemented on a finite precision computer. There needs to be further work on the development of a numerically stable recursive version of the ULLS algorithm. One approach to this problem might be the development of an algorithm based on QR decomposition.

Recursive Levenberg-Marquardt Using Matrix Inversion Lemma

One of the major efforts in this work was the attempt to develop a recursive form of the Levenberg-Marquardt algorithm requiring no matrix inversion. Much progress was made in this area. The most promising recursive Levenberg-Marquardt algorithm developed was a method which uses the matrix inversion lemma once for each diagonal term in the of $\Delta\mu\mathbf{I}$ matrix at each recursion. This implementation proved to be more computationally efficient than any of the standard inversion techniques except Gaussian Elimination. It was shown that if only a portion of the diagonal terms are accounted for at each iteration, the RLMMIL method could achieve great reductions in computational complexity. Because the RLMMIL algorithm is set up to account for the diagonal terms individually, it is possible to adjust only the most significant diagonal terms during a given recursion. If it were possible to "pick" the most significant term or terms at each recursion a reduced set of adjustments made only on the most significant terms could be considered. This would result in a major reduction in the level required computation. In future work, a way of determining which diagonal terms are most significant would need to be developed. This, coupled with the method of making the adjustments shown in this work would comprise a

complete, efficient, recursive training method closely resembling the Levenberg-Marquardt method.

REFERENCES

[1] L. Ljung, *System Identification: Theory for the User*, Prentice Hall, Engelwood Cliffs, N.J.

[2] M. T. Hagan, H. B. Demuth and M. Beale, *Neural Network Design,* Boston: PWS Publishing Co., 1996.

[3] J. M. Mendel, *Lessons in Digital Estimation Theory,* 2nd ed., Prentice-Hall, Englewood CLiffs, N.J., 1995

[4] S. Haykin, *Adaptive Filter Theory*, 3rd ed., Prentice Hall, Upper Saddle River, N.J., 1996.

[5] B. Widrow, M.E. Hoff, "Adaptive switching circuits," *1960 IRE WESCON Convention Record*, New York: IRE Part 4, pp. 96-104, 1960.

[6] R.E. Kalman, "A new approach to linear filtering and prediction problems," *Trans. ASME, J. Basic Eng*., vol. 83, pp. 95-108.

[7] A. H. Sayed and T. Kailath (1994). "A state-space approach to adaptive RLS filtering," *IEEE Signal Process. Mag*., vol. 11, pp. 18-60.

[8] R.E. Kopp, and R. J. Orford. 1963. "Linear regression applied to system identification for adaptive control systems." *AIAA J.*, Vol. 1, p. 2300.

[9] S. Singal and L. Wu, "Training multilayer perceptrons with the extended Kalman algorithm," *Advances in Neural Information Processing Systes 1.* Denver 1988, D. S. Touretzky, Ed. SanMateo, CA: Morgan Kaufmann, 1989, pp. 133-140.

[10] M. B. Mathews, "Neural network nonlinear adaptive filtering using the extended Kalman filter algorithm," *Proceedings of the International Neural Networks Conference, Paris* 1990, vol. I,pp 115-119.

[11] R. J. Williams, "Training recurrent networks using the extended Kalman filter," *International Joint Conference on Neural Networks*, Baltimore 1992, vol. IV, pp. 241-246.

[12] G. V. Puskorius and L. A. Feldkamp, "Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks," IEEE Transactions on Neural Networks, vol. 5, no. 2, pp. 279-297, 1994.

[13] G. V. Puskorius and L. A. Feldkamp, "Recurrent network training with the decoupled extended Kalman filter algorithm," *Proceedings of the 1992 SPIE Conference on the Science of Artificial Neural Networks,* Orlando, 1992, vol. 1710. pp. 461-473

[14] G. V. Puskorius and L. A. Feldkamp, "Automotive engine idle speed control with recurrent neural networks," *Proceedings of the 1993 American Control Conference,* San-Francisco, 1993, pp. 311-316

[15] G. V. Puskorius and L. A. Feldkamp, "Model reference adaptive control with recurrent networks trained by the dynamic DEKF algorithm," *International Joint Conference on Neural Networks*, Baltimore, 1992, vol. II, pp. 106-113

[16] K.S. Narendra, A.M. Parthasrathy, "Identification and Control for Dynamic Systems Using Neural Networks", *IEEE Transactions on Neural Networks*, vol. 1, 1990, pp. 4-27

[17] Medsker, L.R. and Jain, L.C., *Recurrent Neural Network Design and Applications,* CRC Press, New York

[18] A. Bjorck, *Numerical Method for Least Squares Methods,* Siam, 1996.

[19] G.H. Golub and C.F. Van Loan, *Matrix Computations*, 3rd ed., John Hopkins, Baltimore and London

[20] D. P. Bertsekas, "Incremental Least Squares Methods and the Extended Kalman Filter", *SIAM J. on Optimization*, Vol. 6, 1996, pp. 807-822.

[21] O. Stan and E. W. Kamen, "New block recursive MLP training algorithms using the Levenberg-Marquardt algorithm", *International Joint Conference on Neural Networks*, Washington D.C., 1999, vol. III, pp. 1672-1677.

[22] L.S.H. Ngia and J. Sjoberg, "Efficient Training of Neural Nets for Nonlinear Adaptive Filtering Using a Recursive Levenberg-Marquardt Algorithm", *IEEE Transactions on Signal Processing*, vol. 48, 2000, pp. 1915-1927.

[23] W. Yang and M.T. Hagan, "Training recurrent Networks", *Proceedings of the 7th Oklahoma Symposium on Artificial Intelligence*, Stillwater, 226, 1993.

[24] Lua, F.,Unbehaun, R.,Applied Neural Networks for Signal Processing, Cambridge University Press, 1997

[25] K. Ogata, *Discrete-time Control Systems*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[26] Meng Hock Fun, Doctoral Thesis, *Recursive Time and Order Update Algorithms for Radial Basis Function Networks*, Oklahoma State Unversity, 2001

[27] A.P. Christoforou and A.S. Yigit, "Dynamic Modelling of Rotating Drillstrings with Borehole Interactions", *Journal of Sound and Vibration*, Vol. 2, 1997, pp. 243-260.

[28] P.N. Jogi, J.D. Macpherson and M. Neubert, "Field Verification of Model Derived natural Frequencies of a drill String", *Proceedings Energy Sources Technology Conference and Exhibition*, ETCE99-6648, 1999.

[29] G. Heisig, J. Sancho and J.D. Macpherson, "Downhole Diagnosis of Drilling Dynamics Data Provides New Level Drilling Process Control to Driller", *SPE 49206*, September 1998.

[30] J.S. Henneuse, “Surface Detection of Vibrations and Drilling Optimization”, *IADC/SPE 23888*, February 1992.

[31] J.D. Macpherson, P.N. Jogi and J.E.E. Kingman, “Application and Analysis of Simultaneous Near Bit and Surface Dynamics Measurements”, *IADC/SPE 39397*, March 1998.

[32] S.A. Zannoni, C.A. Cheatham, C-K.D. Chen and C.A. Golia, “Development and Field Testing of a New Downhole MWD Drillstring Dynamics Sensor”, *SPE 26341*, October 1993.

VITA

Roger L. Schultz 2

Candidate for the Degree of

Doctor of Philosophy

Thesis: INCREMENTAL TRAINING ALGORITHMS FOR NONLINEAR NEURAL NETWORKS

Major Field: Electrical and Computer Engineering

Biographical:

Personal Data: Born in Stillwater, Oklahoma, on September 28, 1963, the son of R. D. and Emma F. Schultz.

Education: Graduated from Stillwater High School, Stillwater, Oklahoma, in May 1981; received Bachelor of Science degree in Mechanical Engineering and Master of Science in Mechanicl Engineering degree from Oklahoma State University, Stillwater, Oklahoma, in December 1986 and May 1988, respectively. Completed the requirements for the Doctor of Philosophy degree in Electrical and Computer Engineering at Oklahoma State University in December 2002.

Experience: Employed by REN Corporation as a Project Engineer from 1985 to 1988 ; employed by Halliburton Services as a Design Engineer from 1988 to 1989; employed by Halliburton Reservoir Services as Design Engineer from 1989 to 1993; employed by Halliburton Energy Services as a Research Engineer from 1993 to present.

Professional Memberships: Member of Institute of Electrical and Electronic Engineers, and the Society of Petroleum Engineers.