

THE DAMPED NEWTON METHOD

--AN ANN LEARNING

ALGORITHM

By

LIYA WANG


Bachelor of Science
China University of Mining and Technology
XuZhou, P.R. of China
1985

Master of Science
Northern Arizona University
Flagstaff, Arizona
May, 1993


Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
December, 1995

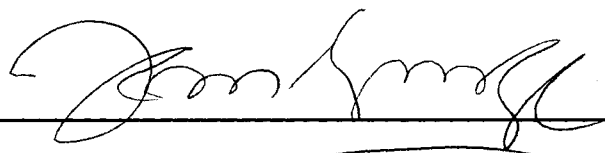
THE DAMPED NEWTON METHOD
--AN ANN LEARNING
ALGORITHM

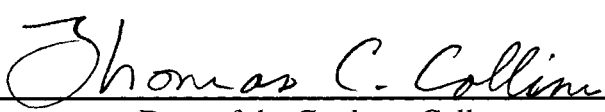
Thesis Approved:



Thesis Advisor







Dean of the Graduate College

PREFACE

This paper presents a new learning algorithm for training fully-connected, feedforward artificial neural networks. The proposed learning algorithm will be suitable for training neural networks to solve approximation problems.

The framework of the new ANN learning algorithm is based on Newton's method for solving non-linear least squares problems. To improve the stability of the new learning algorithm, the Levenberg-Marquardt technique for safe-guarding the Gauss-Newton method is incorporated into the Newton method. This damped version of Newton's method has been implemented using FORTRAN 77, along with some other well-known ANN learning algorithms in order to evaluate the performance of the new learning algorithm. Satisfactory numerical results have been obtained. It is shown that the proposed new learning algorithm has a better performance than the other algorithms in dealing with function approximation problems and problems which may require a high precision of training accuracy.

I would like to express my sincere gratitude to my thesis advisor, Dr. J. P. Chandler, for his guidance in choosing the topic of this thesis and for his thoughtful suggestions and encouragement in pin-pointing and solving problems that I have encountered throughout my thesis work. Special thanks are also due to Dr. K. M. George and Dr. G. E. Hedrick of the Department of Computer Science, who served as the committee members for this

thesis. Their patience in reviewing and revising this report is also much appreciated.

Finally, I would like to thank my parents, Qinghuo Wang and ChuenLing Yang, for rearing me very thoughtfully. My deepest appreciation is extended especially to my mother, who suffered very much in her life and raised my brother and me under some difficult circumstances.

TABLE OF CONTENTS

<i>Chapter</i>	<i>Page</i>
1. INTRODUCTION	1
1.1 ANN History	1
1.2 ANN Models and Applications	2
1.3 Feedforward ANN	3
1.4 Learning in Feedforward ANN	5
1.5 Learning Algorithms for Feedforward ANN	6
1.6 The Purpose of the Paper	8
2. NON-LINEAR OPTIMIZATION	9
2.1 Fundamental Concepts	9
2.2 General Descent	18
2.3 Steepest Descent Method	21
2.4 The Newton Method	23
2.5 Conjugate Gradient Descent Method	26
2.6 Least Squares Minimization ..(Gauss-Newton).....	31
2.7 The Levenberg-Marquardt Method	34
3. ANN LEARNING ALGORITHMS	42
3.1 Architecture of Feedforward ANN	42

3.2 Dynamic Adaptation in Feedforward ANN	48
3.3 Propagated Computations in Feedforward ANN	53
3.4 Classical Learning Algorithms	60
3.5 Implementation of The Levenberg-Marquardt Method	68
4. THE DAMPED NEWTON LEARNING ALGORITHM	71
4.1 The Damped Newton Method	71
4.2 The Damped Newton Learning Algorithm	72
5. IMPLEMENTATION AND TEST RESULTS	75
5.1 Language Implementations	75
5.2 Neural Network Design	78
5.3 Test Problems	80
5.4 Test Results	81
6. CONCLUSION	88
6.1 Conclusion	88
6.2 Recommendation for Future Work	88
7. REFERENCES	90
8. APPENDICES	94
8.1 Program List	94

LIST OF TABLES

<i>Table #</i>		<i>Page</i>
Table 1: 2-parity Test Results	82
Table 2: 3-parity Test Results	83
Table 3: 4-parity Test Results	83
Table 4: Cancer Problem Test Results	84
Table 5: Building Problem Test Results	84
Table 6: Heart Problem Test Results	85
Table 7: GNLM D Performance Test Results on Building Problem	86
Table 8: NLMD Performance Test Results on Building Problem	86
Table 9: GNLM D Performance Test Results on Heart Problem	87
Table 10: NLMD Performance Test Results on Heart Problem	87

LIST OF FIGURES

<i>Chapter.Section.Figure Number</i>	<i>Page</i>
Figure 1.3.1. A simulated neuron	3
Figure 1.3.2. The Sigmoid function	4
Figure 1.3.3. A neural network for XOR classification	4
Figure 3.1.1. Architecture of a three-layer neural network	43
Figure 3.1.2. The Sigmoid function	45
Figure 3.1.3. The hyperbolic tangent function	46
Figure 3.3.1. Part of a neural network	53

1. INTRODUCTION

1.1 ANN History

Artificial Neural Networks (ANNs) emerged in the 1940s when people used a single threshold device to model the functionality of a biological neuron [20]. Simulating the network structure of the human brain, those artificial neurons were linked together to form a network (called artificial neural network, neural network, or neural net, etc.). While a human brain is characterized by its learning capability, the learning capacity of the linked neurons was first shown in the book titled “*The Organization of Behavior*” by Donald Hebb, published around 1950 [18]. Two early successes in the 1950s and 1960s were Rosenblatt’s Perceptron and Widrow’s ADALINE (ADaptive LINear Element), equipped with the Perceptron learning law and Widrow’s learning law respectively [18]. Both were able to learn and to perform accordingly. Despite some setbacks in the late 1960s and 1970s, the ANN researches regained their strength in the 1980s due to the contributions of many dedicated scholars. The publication of the famous Error Back-Propagation Algorithm (Backpropagation, Backprop, or BB Algorithm) by Rumelhart, Hinton, and Williams in 1986 (also independently done by Paul Werbos in 1974 [30]) set the milestone for the ANN’s prosperity beginning in the late 1980s [15]. Today, the studies of ANNs have become an independent and very broad field, and involve the endeavors of thousands of scholars and engineers in many different fields all over the world.

1.2 ANN Models and Applications

An artificial neural network is an information processing system [10]. It usually consists of a large number of artificial neurons. These neurons are linked together through direct connections to form a network. Currently there are several such network models: feedforward network model, feedback (recurrent) network model, and cellular model. One example of the feedforward network is the fully-connected feedforward neural network. Such networks are very useful in dealing with classification problems and function approximation problems. The Hopfield network and the Boltzmann machine are examples of the recurrent models. They are often used in speech processing and pattern recognition [20]. The Kohonen map is a form of the cellular model. Such network is self-organizing and is mostly used in speech recognition [20].

An ANN can be considered to be a parallel machine. It distributes its computing power among all neurons in the network. Each neuron, a simple local processing unit, contributes to the final output of the network. This characteristic of the ANN is very helpful to the application problems that require massive and high-speed data processing capacities.

Today, ANNs have been used in many fields. Such fields include data encoding and compression, signal and image processing, speech recognition, pattern classification, noise filtering, stock market predictions, credit card application processing, and modeling [2]. Currently, there are several kinds of neural network chips and hundreds of software packages available for developing neural network applications. Thousands of neural network applications have been carried out successfully and more are being explored in new areas and in new fields.

1.3 Feedforward ANN

In this paper, we concentrate on one particular ANN model: the fully-connected, feedforward neural network. In the sections that follow, all neural networks mentioned will refer to the fully-connected, feedforward neural networks unless otherwise stated.

A fundamental feature of the feedforward neural network is its adaptive behavior, i.e., the capability to learn. When a feedforward ANN is employed to tackle a problem, all that are needed is some training examples of the problem, i.e., some input-output patterns. Based on those sample patterns, the ANN, when properly set up, not only tries to learn how to handle those samples correctly, but also summarizes the learning results and tries to handle samples in the entire problem domain. This capability of the ANNs is especially useful to solve the kind of problems whose underlying ideas are not clear and/or there are no fast rules that can easily be applied.

The basic building blocks of a feedforward neural network are the artificial neurons and the connections. A connection usually has a weight on it to represent the level of importance of (or contribution from) that connection, except perhaps for the output connections of the network. A neuron (Figure 1.3.1) is a basic processing unit: it sums up all its inputs, modulates each by its corresponding connection weight, and then transforms the result via some activation function to yield the output of the neuron.

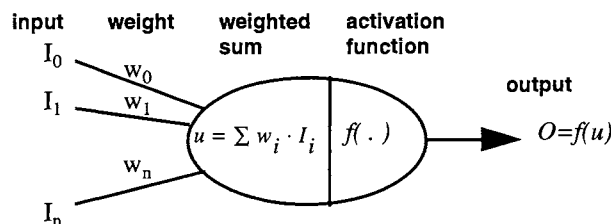


Figure 1.3.1: A simulated neuron.

The type of the activation function determines the sort of problems that an ANN can solve. For example, ANNs with step (threshold) activation functions can only solve classification problems. In order to approximate non-linear mappings, non-linear activation functions have to be used. In this paper, the non-linear sigmoidal activation function will be used (Figure 1.3.2).

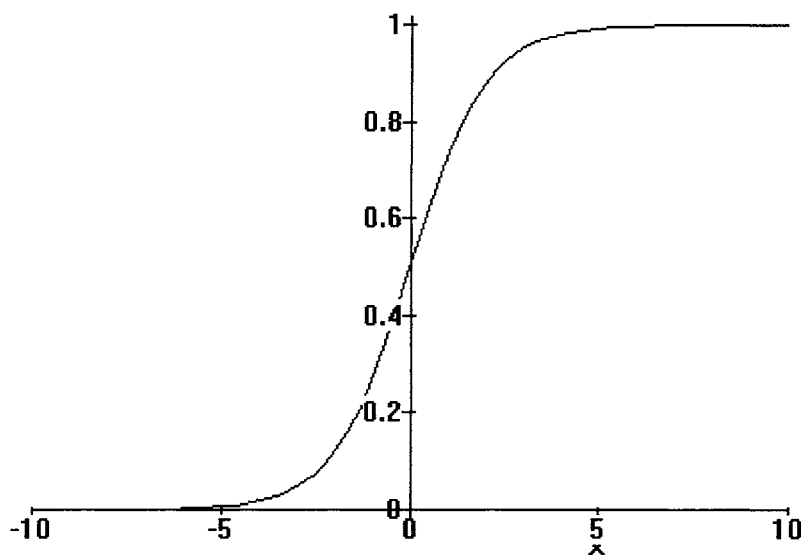


Figure 1.3.2: The Sigmoid function

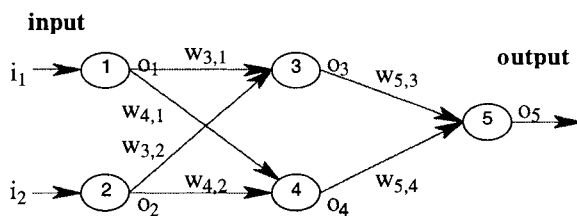


Figure 1.3.3: a neural network for XOR classification

All neurons in a feedforward network are organized into layers or slabs (Figure 1.3.3). The outputs of the neurons in one layer (or the inputs of the network) are all linked only to each of the neurons in the next layer, except for the output layer of the network. A neural network connected this way is called a fully-connected neural network. When input signals are fed in, the computation of the network is carried out on a layer-by-layer basis, starting at the input layer. This processing pattern continues until the outputs of the network have been produced. Such a computation process is called a forward pass, which is a left-to-right pass as shown in a network layout.

1.4 Learning in Feedforward ANN

Learning is an intrinsic requirement of neural networks--an untrained network with randomly assigned connection weights can do nothing meaningful. The knowledge representation system employed by an ANN is opaque: the ANN has to learn its own representation because programming it by hand is not possible. That is, a neural net has to be trained in order to perform a defined task. Thus, the use of a neural network usually involves two major stages: the learning stage and the performing stage. During the learning stage, a neural network produces output for each input sample and the result is compared with the required output. If a mistake is made or the approximation result is not desirable, then the net tries to learn from this example and modifies its behavior in order to make fewer mistakes or better approximation results. Such a way of training is called supervised training or learning with a teacher. After perhaps hundreds, sometimes even up to hundreds of thousands of round of repetitions as the net goes through all the training examples, the trained network will enter the performing stage. In this stage, the ANN is

used, based upon its generalization of the training examples, to compute outputs for other non-example inputs.

While the learning of a neural network is observed as a change of behavior, inside the neural network, learning takes place in the form of weight changing. Usually, some sort of optimization function is used to measure the output errors. Then, a learning algorithm is applied to transform the measured error information into weight changes. The major learning algorithms that are used for training the feedforward nets are those that enforce the learning process by means of backpropagation. Those learning algorithms are discussed in the next section.

1.5 Learning Algorithms for Feedforward ANN

A learning law governs how a neural network enforces learning. The following is a general outline of the learning procedure used in all of the backpropagation algorithms:

- The network usually starts out with a random set of weights.
- Examples (input-output pairs) are presented at the input side of the network.
- Each input-output pair requires two stages: a forward pass and a backward pass. The forward pass is to present a sample input to the network and to let activation flow until they reach the output layer.
- During the backward pass, the network's actual output from the forward pass is compared with the desired output and error estimates are computed for the output nodes. Then the network adjusts its weights in a backward fashion, starting at the output layer (or saves all adjustments and changes weights after all examples have been presented), in order to reduce the errors.

- The process is repeated many times until some error criteria are met.

Several mathematical models in optimization theory can be used in carrying out the learning process of the above procedure. In that regard, the learning process corresponds to the minimization process of the models. Among the available optimization models, the Least Squares model is the one used the most in ANNs. Suppose that n , l , and m are the dimensions of the input vector, weight vector, and output vector respectively and s is the number of training examples. For each $i=1, \dots, m$, let $Y_i(\mathbf{x}, \mathbf{w})$ be the i -th coordinate in the output vector. Then, the least squares model for training an ANN is

$$(1.5.1) \quad E(\mathbf{w}) = \sum_{j=1}^s \left(\sum_{i=1}^m (Y_i(\mathbf{x}_j, \mathbf{w}) - y_{j,i})^2 \right),$$

where \mathbf{x}_j is the j -th input vector, and $y_{j,i}$ is the i -th coordinate of the output vector \mathbf{y}_j , corresponding to the input \mathbf{x}_j .

In the above optimization model, learning corresponds to minimizing $E(\mathbf{w})$ with regard to the weight vector \mathbf{w} . There are many ways to achieve this optimization goal and it is the mathematical methods used in the algorithms that distinguish one from the others. Currently, most of the gradient-based backpropagation learning algorithms can be traced to three basic mathematical methods, which are well established in the optimization theory. They are the Steepest Descent method, the Conjugate Gradient method, and the Newton method. Many learning algorithms have been successfully built upon those methods and more modified versions have come out in order to improve the performance of the original versions. One remarkable method that has been used in some modified algorithms is the Levenberg-Marquardt method [24], which is a modification of the Gauss-Newton method (derived from the Newton method). In this paper, we develop a

new ANN learning algorithm that will be based on the framework of the Levenberg-Marquardt method. The goal of the paper is stated in the next section.

1.6 The Purpose of the Paper

The aim of this paper is to introduce a new supervised learning algorithm for training fully-connected, feedforward neural networks. The proposed algorithm is based on the well-known Newton numerical approximation method and the Levenberg-Marquardt improvement of the Gauss-Newton method. Specifically, we incorporate Newton's method with the technique used in the Levenberg-Marquardt method to derive an improved version of the Newton method.

To investigate further into the subject, we first need some background knowledge. In theory, there is considerable overlap between the fields of neural networks and statistics. Many well-known results from the statistical theory of non-linear models, as we shall see in the next few chapters, can be applied directly to feedforward neural networks. The next Chapter is devoted to a brief introduction of non-linear optimization theory.

2. NON-LINEAR OPTIMIZATION

In this chapter, we will introduce some of the classical minimization methods. Although, the approaches to derive these methods are different, they have two things in common. First, all these methods are iterative. That is to say, to locate a minimizer \mathbf{x}^* of a function f , a sequence $\{\mathbf{x}^{(k)}\}$ of points is generated. If the sequence converges, then an estimate of \mathbf{x}^* to a satisfactory degree of accuracy has been attained. Second, all these methods are descent methods that contain the following two fundamental steps at each iteration.

- 1) Determine a search direction along which a reduction of f in function values is possible.
- 2) Choose a step size so that minimization does take place.

We shall start, in section 2.2, with a discussion of a general descent method consisting of the above two basic steps and then study the classical ones in the sections that follow. The first section below provides some of the background knowledge we will need for the discussion of later sections.

2.1 Fundamental Concepts

Let \mathfrak{R} be the set of all real numbers.

Linear and matrix algebra

An $m \times n$ matrix $\mathbf{A} \in \mathfrak{R}^{m \times n}$ given by

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}_{m \times n}$$

can be expressed compactly as $\mathbf{A} = [a_{ij}]_{m \times n}$. It has m rows and n columns. An $m \times 1$ matrix will be a column *vector* of dimension m and a $1 \times n$ matrix will be called a row *vector* of dimension n . In our discussion, matrices are denoted by uppercase bold face letters and vectors by lowercase bold face letters.

The *transpose* of any matrix is obtained by rewriting all columns as rows. Thus, the transpose of an $m \times n$ matrix are an $n \times m$ matrix. In symbol, the transpose of a matrix \mathbf{A} is denoted by \mathbf{A}^T . Note that for any matrix \mathbf{A} , we have

$$(\mathbf{A}^T)^T = \mathbf{A}$$

and

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T,$$

where $\mathbf{A} \in \mathfrak{R}^{m \times n}$, $\mathbf{B} \in \mathfrak{R}^{n \times l}$, and $\mathbf{AB} = \left[\sum_{k=1}^n a_{ik} b_{kj} \right]_{m \times l}$.

An $n \times n$ matrix is called a *square* matrix. A *symmetric* matrix is a square matrix \mathbf{A} such that

$$\mathbf{A} = \mathbf{A}^T$$

A *diagonal* matrix $\mathbf{D} \in \mathfrak{R}^{n \times n}$ is a square matrix whose entries $d_{ij} = 0$, for all $i \neq j$, and is written as

$$\mathbf{D} = \text{diag}(d_{11}, \dots, d_{nn}).$$

A diagonal matrix is called the *identity* matrix if all of its diagonal elements are equal to 1. An identity matrix is often denoted by \mathbf{I} .

A square matrix \mathbf{B} is said to be the *inverse* of another square matrix \mathbf{A} if

$$\mathbf{AB}=\mathbf{I}.$$

In this case, \mathbf{B} can be written as $\mathbf{B}=\mathbf{A}^{-1}$, and it could be shown that $\mathbf{BA}=\mathbf{I}$. A square matrix \mathbf{A} is a *singular* matrix if its inverse does not exist. Otherwise, it is said to be *non-singular*.

A symmetric matrix $\mathbf{A}\in\mathfrak{R}^{n\times n}$ is said to be *positive definite* (*negative definite*) if the quadratic form $\mathbf{x}^T\mathbf{A}\mathbf{x}$ satisfies

$$\mathbf{x}^T\mathbf{A}\mathbf{x} > 0 \text{ (<0)}, \text{ for all } \mathbf{x}\neq\mathbf{0} \text{ and } \mathbf{x}\in\mathfrak{R}^n.$$

A matrix \mathbf{A} is positive (or negative) *semidefinite* if the equality sign is included in the condition above. Note that the matrices $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ are always semidefinite for any \mathbf{A} . If a matrix \mathbf{A} is positive or negative definite, then its inverse matrix exists.

A system of linear equations, given by

$$\begin{cases} a_{11}x_1 + \dots + a_{1n} = b_1 \\ \vdots \\ a_{m1}x_1 + \dots + a_{mn} = b_m \end{cases},$$

can be written in matrix form as

$$\mathbf{Ax}=\mathbf{b},$$

where $\mathbf{A}=[a_{ij}]_{m\times n}$, $\mathbf{x}=(x_i)^T$, and $\mathbf{b}=(b_i)^T$. The system is *solvable* if the inverse matrix of \mathbf{A} exists. In case \mathbf{A}^{-1} exists, the *solution* of the system can be written as

$$\mathbf{x}=\mathbf{A}^{-1}\mathbf{b}.$$

To solve a system of linear equations, we need an algorithmic procedure. The algorithm we shall use later on is based on the well-known Gaussian elimination method. To reduce the round-off error associated with finite-digit arithmetic, a technique called

maximal column pivoting or partial pivoting is incorporated into the algorithm [9], which is listed below.

Algorithm 2.1.1: given an $n \times n$ system of linear equations $\mathbf{A}_{n,n} \mathbf{x} = \mathbf{b}$, with $\mathbf{A}_{n,n} = (a_{ij})$ the coefficient matrix.

1. set $a(i, j) = a_{ij}$, all i, j , and For $i = 1, \dots, n$, set $NROW(i) = i$.
2. For $i = 1, \dots, n-1$, repeat step 3, 4, 5, and 6.
3. Let p be the smallest integer with $i \leq p \leq n$, and

$$|a(NROW(p), i)| = \max_{i \leq j \leq n} |a(NROW(j), i)|.$$

4. If $a(NROW(p), i) = 0$, then print "no unique solution exists" and stop.
5. If $NROW(i) \neq NROW(p)$, then set

$$NCOPY = NROW(i), NROW(i) = NROW(p), NROW(p) = NCOPY.$$

6. For $j = i+1, \dots, n$, do 6.1 and 6.2

$$6.1. \text{ set } m(NROW(j), i) = \frac{a(NROW(j), i)}{a(NROW(i), i)}.$$

$$6.2. \text{ Let } E_{NROW(j)} = E_{NROW(j)} - m(NROW(j), i) E_{NROW(i)}.$$

7. If $a(NROW(n), n) = 0$, then print "no unique solution exists" and stop.
8. Otherwise, set

$$x_n = \frac{a(NROW(n), n+1)}{a(NROW(n), n)}.$$

9. For $i = n-1, \dots, 1$, set

$$x_i = \frac{a(NROW(i), n+1) - \sum_{j=i+1}^n a(NROW(i), j) \cdot x_j}{a(NROW(i), i)}.$$

10. Output the solution $\mathbf{x} = (x_1, \dots, x_n)$.

The *inner product* of two real numbered n -dimensional vectors $\mathbf{x} = (x_i)$ and $\mathbf{w} = (w_i)$ is defined as

$$\langle \mathbf{x}, \mathbf{w} \rangle = \mathbf{x}^T \mathbf{w} = \mathbf{w}^T \mathbf{x} = \sum_{i=1}^n w_i x_i.$$

A p -norm (or L_p -norm) of a vector $\mathbf{x}=(x_i)_{1,n}$ is given by

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^n |x_i|^p \right)^{1/p} .$$

A frequently used norm is the 2-norm, also called the *Euclidean* norm. In 2 or 3 dimensional real spaces, the 2-norm of a vector is just the length of the vector.

In a Euclidean normed vector space, we have *Schwarz's* inequality (Cauchy-Schwarz)

$$|\langle \mathbf{x}, \mathbf{y} \rangle| = |\mathbf{x}^T \mathbf{y}| \leq \|\mathbf{x}\|_2^{1/2} \|\mathbf{y}\|_2^{1/2}, \text{ with}$$

equality holding if and only if $\mathbf{x}=\lambda\mathbf{y}$, for some $\lambda \in \mathfrak{R}$.

Multivariate analysis

A set $D \subseteq \mathfrak{R}^n$ is a *convex* set if whenever $\mathbf{x}_1, \mathbf{x}_2 \in D$, the line segment

$$[\mathbf{x}_1, \mathbf{x}_2] = \{ \mathbf{y} \in \mathfrak{R}^n \mid \mathbf{y} = (1-\lambda) \mathbf{x}_1 + \lambda \mathbf{x}_2, 0 \leq \lambda \leq 1 \}$$

lies entirely in D .

A real-valued function $f : \mathfrak{R}^n \rightarrow \mathfrak{R}$ is continuous at $\mathbf{x} \in \mathfrak{R}^n$, if any sequence $\{\mathbf{x}^{(k)}\}$ such that

$$\mathbf{x}^{(k)} \rightarrow \mathbf{x}, \text{ as } k \rightarrow \infty, \text{ then } f(\mathbf{x}^{(k)}) \rightarrow f(\mathbf{x}), \text{ as } k \rightarrow \infty.$$

A function f is said to be continuous on \mathfrak{R}^n , if f is continuous at each $\mathbf{x} \in \mathfrak{R}^n$.

A real-valued vector function $\mathbf{F} : \mathfrak{R}^n \rightarrow \mathfrak{R}^m$, $\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), \dots, f_m(\mathbf{x}))^T$, is continuous on \mathfrak{R}^n , if each f_i is continuous on \mathfrak{R}^n , $i=1, \dots, m$.

Let $f : \mathfrak{R}^n \rightarrow \mathfrak{R}$ be a twice differentiable function. The first partial derivatives of f with regard to (w.r.t.) $x_i, i=1, \dots, n$, is denoted by

$$\partial_i f(\mathbf{x}) := \frac{\partial f(\mathbf{x})}{\partial x_i}, \quad i=1, \dots, n,$$

and the second partial derivative is written as

$$\partial_i \partial_j f(\mathbf{x}) := \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}, \quad i, j=1, \dots, n.$$

The *gradient* $\mathbf{g}(\mathbf{x})$ or $\nabla f(\mathbf{x})$ of f is defined upon all the first partials of f given by

$$\mathbf{g}(\mathbf{x}) := \nabla f(\mathbf{x}) := \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = (\partial_1 f(\mathbf{x}), \dots, \partial_n f(\mathbf{x}))^T.$$

The *Hessian* matrix $\mathbf{H}(\mathbf{x})$ or $\nabla^2 f(\mathbf{x})$ of the function f is defined to be

$$\mathbf{H}(\mathbf{x}) := \nabla^2 f(\mathbf{x}) := \frac{\partial}{\partial \mathbf{x}} \left[\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right]^T = \begin{bmatrix} \partial_1^2 f(\mathbf{x}) & \partial_1 \partial_2 f(\mathbf{x}) & \cdots & \partial_1 \partial_n f(\mathbf{x}) \\ \partial_2 \partial_1 f(\mathbf{x}) & \partial_2^2 f(\mathbf{x}) & \cdots & \partial_2 \partial_n f(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \partial_n \partial_1 f(\mathbf{x}) & \partial_n \partial_2 f(\mathbf{x}) & \cdots & \partial_n^2 f(\mathbf{x}) \end{bmatrix}_{n \times n}.$$

Let $\mathbf{F}: \mathcal{R}^n \rightarrow \mathcal{R}^m$, $\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), \dots, f_m(\mathbf{x}))^T$, be a vector function (written in bold face).

Then, The first order derivative of \mathbf{F} w.r.t. \mathbf{x} is defined by

$$\nabla \mathbf{F}(\mathbf{x}) = (\nabla f_1(\mathbf{x}), \dots, \nabla f_m(\mathbf{x})).$$

Note that $\nabla f_i(\mathbf{x}) = (\partial_1 f_i(\mathbf{x}), \dots, \partial_n f_i(\mathbf{x}))^T$. Hence $\nabla \mathbf{F}(\mathbf{x})$ is a $n \times m$ matrix whose transpose is defined to be the *Jacobian* matrix $\mathbf{J}(\mathbf{x})$ of $\mathbf{F}(\mathbf{x})$, i.e.,

$$\mathbf{J}(\mathbf{x}) := \frac{\partial \mathbf{F}(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \partial_1 f_1(\mathbf{x}) & \partial_2 f_1(\mathbf{x}) & \cdots & \partial_n f_1(\mathbf{x}) \\ \partial_1 f_2(\mathbf{x}) & \partial_2 f_2(\mathbf{x}) & \cdots & \partial_n f_2(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \partial_1 f_m(\mathbf{x}) & \partial_2 f_m(\mathbf{x}) & \cdots & \partial_n f_m(\mathbf{x}) \end{bmatrix}_{m \times n}.$$

The following formulas [10] are useful for differentiating scalar functions with respect to a vector.

$$(2.1.1) \quad \frac{\partial}{\partial \mathbf{x}} (\mathbf{F}(\mathbf{x})^T \mathbf{G}(\mathbf{x})) = \left[\frac{\partial \mathbf{F}(\mathbf{x})}{\partial \mathbf{x}} \right]^T \mathbf{G}(\mathbf{x}) + \left[\frac{\partial \mathbf{G}(\mathbf{x})}{\partial \mathbf{x}} \right]^T \mathbf{F}(\mathbf{x}), \text{ where } \mathbf{F}(\mathbf{x}) \text{ and } \mathbf{G}(\mathbf{x})$$

are vector functions.

$$(2.1.2) \quad \frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{y}) = \mathbf{y},$$

$$(2.1.3) \quad \frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{x}) = 2\mathbf{x},$$

$$(2.1.4) \quad \frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{y}) = \mathbf{A} \mathbf{y},$$

$$(2.1.5) \quad \frac{\partial}{\partial \mathbf{x}} (\mathbf{y}^T \mathbf{A} \mathbf{x}) = (\mathbf{y}^T \mathbf{A})^T = \mathbf{A}^T \mathbf{y},$$

$$(2.1.6) \quad \frac{\partial}{\partial \mathbf{x}} (\mathbf{x}^T \mathbf{A} \mathbf{x}) = (\mathbf{A} + \mathbf{A}^T) \mathbf{x},$$

Let $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ be a function of $\mathbf{x}=(x_1, \dots, x_n)$, having continuous partial derivatives.

Assume $x_i=h(t)$ is differentiable for all $i=1, \dots, n$. Then, the Chain rule of differentiation is

$$\frac{\partial f(t)}{\partial t} = \sum_{i=1}^n \frac{\partial f(\mathbf{x})}{\partial x_i} \frac{\partial x_i(t)}{\partial t}.$$

Taylor Theorem: Let $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ have continuous first (second) partial derivatives and $\mathbf{x}^* \in \mathfrak{R}$. Then, for any \mathbf{x} near \mathbf{x}^* , there exists $\theta \in (0, 1)$ such that

$$f(\mathbf{x}) = f(\mathbf{x}^*) + \mathbf{g}(\mathbf{y})^T (\mathbf{x} - \mathbf{x}^*)$$

$$\left(f(\mathbf{x}) = f(\mathbf{x}^*) + \mathbf{g}(\mathbf{x})^T (\mathbf{x} - \mathbf{x}^*) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^*)^T \mathbf{H}(\mathbf{y}) (\mathbf{x} - \mathbf{x}^*) \right),$$

where, $\mathbf{y}=\mathbf{x}^* + \theta(\mathbf{x}-\mathbf{x}^*)$, \mathbf{g} is the gradient vector of f and \mathbf{H} is the Hessian matrix of f .

A *critical* point of a function $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$, having continuous first partial derivatives, is a point \mathbf{x}^* in the domain of f at where

$$\mathbf{g}(\mathbf{x}^*)=0,$$

where \mathbf{g} is the gradient of f . The point \mathbf{x}^* is a *strong global minimizer* of f if

$$f(\mathbf{x}) > f(\mathbf{x}^*), \text{ for all } \mathbf{x} \text{ in the domain of } f.$$

The point \mathbf{x}^* is a *strong local minimizer* of f if the above condition holds on a convex subset of the domain of f . Note that a minimizer is necessarily a critical point. For a critical point to be a strong local minimizer of f , we have the following theorem [36].

Theorem 2.1.2: If 1. $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ has continuous first and second partial derivatives in an open convex set D containing \mathbf{x}^* ;

2. \mathbf{x}^* is a critical point of f in D ;

3. $\mathbf{H}(\mathbf{x}^*)$ is positive definite, where \mathbf{H} is the Hessian matrix of f ,

then, \mathbf{x}^* is a strong local minimizer of f over D .

line search

The minimization of a univariate function of the form $\phi(\alpha) = f(x + \alpha p)$ over an interval $(a, b) \subset \mathfrak{R}$ requires the uses of line search techniques. The commonly used such line search techniques are the *Golden Section* search method, *Bisection* method, *Quadratic Interpolation* method, and *Cubic Interpolation* method. The following algorithm [28] is based on the quadratic interpolation method, which we shall use later.

Algorithm 2.1.2: Quadratic Interpolation: given an initial interval (a_1, b_1) , a point $c_1 \in (a_1, b_1)$, and *tolerance*.

1. set $f_a = f(a_1)$, $f_b = f(b_1)$, $f_c = f(c_1)$.

2. set $k=1$.

3. set
$$\hat{x}^* = \frac{1}{2} \frac{a_k^2(f_c - f_b) + b_k^2(f_a - f_c) + c_k^2(f_b - f_a)}{a_k(f_c - f_b) + b_k(f_a - f_c) + c_k(f_b - f_a)}$$

(Note: if denominator in the above formula is 0, then stop.)

$$f_x = f(\hat{x}^*)$$

if $\hat{x}^* < c_k$ and $f_x < f_c$

then set $a_{k+1} = a_k$, $b_{k+1} = c_k$, $c_{k+1} = \hat{x}^*$

$$f_b = f_c, f_c = f_x$$

else if $\hat{x}^* > c_k$ and $f_x > f_c$

then set $a_{k+1} = a_k$, $b_{k+1} = \hat{x}^*$, $c_{k+1} = c_k$.

$$f_b = f_x$$

else if $\hat{x}^* < c_k$ and $f_x > f_c$

then set $a_{k+1} = \hat{x}^*$, $b_{k+1} = b_k$, $c_{k+1} = c_k$.

$$f_a = f_x$$

else set $a_{k+1} = c_k$, $b_{k+1} = b_k$, $c_{k+1} = \hat{x}^*$,

$$d_{k+1} = b_{k+1} - (1-\lambda)(b_{k+1} - a_{k+1}),$$

$$f_a = f_c, f_c = f_x$$

4. if $b_{k+1} - a_{k+1} < \textit{tolerance}$, or $(f(c_k) - f(c_{k+1})) / f(c_k) < \textit{tolerance}$, then stop

5. otherwise, set $k = k + 1$, go to 3.

In the above algorithms, it is assumed that a minimizer lies in the given initial interval.

If this is not available, then methods have to be incorporated into the algorithms to locate such an interval first. The methods that are commonly in use to find such an interval are *the function comparison* method and *extrapolation* method, which are listed in [28].

Rate of convergence

If a minimization method for minimizing a function f generates a convergent sequence $\{\mathbf{x}^{(k)}\}$ that approaches a minimizer \mathbf{x}^* of f , we then are interested in the speed of convergence of the algorithm. An algorithm is said to give *p-th order rate of convergence* if p is the largest number such that the limit

$$P = \lim_{k \rightarrow \infty} \frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\|_2}{\|\mathbf{x}^{(k)} - \mathbf{x}^*\|_2^P}$$

exists. An algorithm with first-order rate of convergence is called an algorithm of linear convergence and if, in addition, $P=0$, then it is said to be of superlinear convergence.

Having covered some of the basic knowledge that we shall need in later sections, we now start with the introduction of a basic minimization technique--the general descent method.

2.2 General Descent

In this section, we discuss a general descent scheme for minimizing a function, which forms the skeleton of all the algorithms we shall introduce in this chapter. We need the following definition to start with [36].

Definition 2.2.1: Assume $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ has first partial derivatives at a point \mathbf{x}^* , and let $\mathbf{p} \in \mathfrak{R}^n$ be a non-zero vector. Then, \mathbf{p} is *downhill* for f at \mathbf{x}^* if and only if $\mathbf{g}(\mathbf{x}^*)^T \mathbf{p} < 0$, where \mathbf{g} is the gradient of f .

Let function $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ have continuous second partial derivatives at a point $\mathbf{x}_0 \in \mathfrak{R}^n$, then by Taylor's theorem, we have

$$(2.2.1) \quad f(\mathbf{x}_0 + \alpha \mathbf{p}) = f(\mathbf{x}_0) + \alpha \mathbf{g}(\mathbf{x}_0)^T \mathbf{p} + \frac{1}{2} \alpha^2 \mathbf{p}^T \mathbf{G}(\mathbf{y}) \mathbf{p},$$

where $\mathbf{y} = \mathbf{x}_0 + \theta \alpha \mathbf{p}$, for some $\theta \in (0, 1)$. It is not hard to show that as $\alpha \rightarrow 0$, the sign of the last two terms in (2.2.1) would be dominated by the sign of the term $\alpha \mathbf{g}(\mathbf{x}_0)^T \mathbf{p}$ [36].

Now, assuming that \mathbf{p} is downhill at \mathbf{x}_0 , i.e., $\mathbf{g}(\mathbf{x}_0)^T \mathbf{p} < 0$, we then have, for α sufficiently small,

$$f(\mathbf{x}_0 + \alpha \mathbf{p}) < f(\mathbf{x}_0).$$

This implies that the function value of f would be reduced if we take an appropriate step along a downhill direction. Hence, starting with an initial point $\mathbf{x}^{(0)}$, we could use the Taylor expansion of f around $\mathbf{x}^{(0)}$ to find another point, say $\mathbf{x}^{(1)}$, which results in the function-value reduction of f , and then, successively repeat the processes for the newly derived point $\mathbf{x}^{(k)}$, $k=1, 2, \dots$. This iterative process will generate a sequence $\{\mathbf{x}^{(k)}\}$ that results in the successive reduction of f in function values. Such consideration yields the following *general descent* algorithmic scheme [36]:

Algorithm 2.2.1: Let an estimate $\mathbf{x}^{(0)}$ of an unconstrained minimizer \mathbf{x}^* of f be given

1. Set $k=0$.
2. Compute $\mathbf{p}^{(k)}$ such that $\mathbf{g}(\mathbf{x}^{(k)})^T \mathbf{p}^{(k)} < 0$.
3. Compute $\alpha^{(k)}$ such that $f(\mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}) < f(\mathbf{x}^{(k)})$.
4. Compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$.
5. If $\mathbf{x}^{(k+1)}$ satisfies given convergence criteria, then stop.
6. Set $k=k+1$, go to 2.

In *Algorithm 2.2.1*, it is readily seen that Step 2 and 3 are the two fundamental procedures that we have outlined in section 2.1, though detailed methods as how to carry out those two steps are not given.

Clearly, there are many choices of $\alpha^{(k)}$ and $\mathbf{p}^{(k)}$ that satisfy the criteria in Step 2 and 3 of *Algorithm 2.2.1*. However, the choices of $\alpha^{(k)}$ and $\mathbf{p}^{(k)}$ alone are not sufficient to ensure convergence of $\{\mathbf{x}^{(k)}\}$ to a minimizer of f . The conditions to ensure that $\{\mathbf{x}^{(k)}\}$ will converge to a minimizer of f can be found in [36].

A frequently used technique to determine $\alpha^{(k)}$ in Step 3 above is to estimate a local minimizer of $f(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)})$ which is regarded as a function of α . Then, at least approximately, $\alpha^{(k)}$ would satisfy the requirement

$$(2.2.2) \quad f(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)}) = \min_{\alpha} (f(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)})).$$

Let $\phi: \mathfrak{R} \rightarrow \mathfrak{R}$ be defined by

$$(2.2.3) \quad \phi(\alpha) = f(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)}).$$

Then (2.2.2) is equivalent to finding a local minimizer $\alpha^{[k]}$ of ϕ , so that, at least approximately,

$$(2.2.4) \quad \phi'(\alpha^{[k]}) = 0,$$

where ϕ' is the first derivative of ϕ . Normally, (2.2.4) is a non-linear equation and can not be solved analytically. Numerically, this can be done by using one of the line search techniques mentioned in section 2.1. Hereafter, for all the algorithms we will introduce, we shall use (2.2.2) as the criterion for computing $\alpha^{[k]}$ at each iteration.

An important consequence of using (2.2.2) is an equation that shows the relation between the search direction $\mathbf{p}^{(k)}$ and the gradient $\mathbf{g}^{(k+1)}$ of f at $\mathbf{x}^{(k+1)}$. We have

$$\begin{aligned} \phi'(\alpha) &= \frac{d}{d\alpha} f(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)}) \\ &= \sum_{i=1}^n \frac{\partial}{\partial x_i} f(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)}) \frac{\partial}{\partial \alpha} (\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)}) \\ &= \mathbf{g}(\mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)})^T \mathbf{p}^{(k)}. \end{aligned}$$

By (2.2.4), $\phi'(\alpha^{(k)}) = 0$, which implies that

$$(2.2.5) \quad \mathbf{g}^{(k+1)T} \mathbf{p}^{(k)} = \phi'(\alpha^{(k)}) = 0.$$

The geometric meaning of (2.2.5) is that the two vectors \mathbf{p} and \mathbf{g} are orthogonal.

Many methods have been introduced for the determination of the $\mathbf{p}^{(k)}$'s in *Algorithm*

2.2.1. The following few sections will discuss some of them in detail.

2.3 Steepest Descent Method

Suppose that the gradient vector \mathbf{g} of a given function f can be calculated analytically, then we can choose for the vector \mathbf{p} in *Algorithm 2.2.1* as $-\mathbf{g}$. That is, the descent direction $\mathbf{p}^{(k)}$ at each iteration will be $-\mathbf{g}^{(k)} := -\mathbf{g}(\mathbf{x}^{(k)}) = -\nabla(f)_{\mathbf{x}^{(k)}}$.

Assume $\mathbf{g}^{(k)} \neq 0$, we have

$$(2.3.1) \quad \mathbf{p}^{(k)T} \mathbf{g}^{(k)} = -\mathbf{g}^{(k)T} \mathbf{g}^{(k)} = -\sum_i (g_i^{(k)})^2 < 0.$$

Hence, if we choose the search direction $\mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$, then $\mathbf{p}^{(k)}$ is downhill for f at $\mathbf{x}^{(k)}$ for each integer $k > 0$.

Note that since, by Taylor's theorem, for $\alpha^{(k)}$ sufficient small, we have a truncated first order Taylor representation of f that yields the following approximation.

$$(2.3.2) \quad f(\mathbf{x}^{(k+1)}) - f(\mathbf{x}^{(k)}) = f(\mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}) - f(\mathbf{x}^{(k)}) \cong \alpha^{(k)} \mathbf{g}^{(k)T} \mathbf{p}^{(k)}.$$

In (2.3.2), we can see that the reduction of f in function values at each iteration depends approximately on the magnitude of $\mathbf{p}^{(k)T} \mathbf{g}^{(k)}$, which, by Schwarz' inequality, is bounded by the product of the 2-norms (Euclidean norm) of the 2 factors, i.e.,

$$(2.3.3) \quad \left| \mathbf{p}^{(k)T} \mathbf{g}^{(k)} \right| \leq \|\mathbf{p}^{(k)}\|_2 \|\mathbf{g}^{(k)}\|_2,$$

with equality holds if and only if $\mathbf{p}^{(k)} = \lambda \mathbf{g}^{(k)}$ ($\lambda \in \mathfrak{R}$). Hence, if we take $\mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$, then

$\mathbf{p}^{(k)T} \mathbf{g}^{(k)}$ has maximum magnitude. This would result in approximately the maximum

reduction of function values of f at each iteration. Such a consideration justifies the name steepest descent method for the descent method obtained via taking $\mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$ in *Algorithm 2.2.1*. The steepest descent method is contained in the following algorithm [36].

Algorithm 2.3.1: Let an estimate $\mathbf{x}^{(0)}$ of an unconstrained minimizer \mathbf{x}^* of f be given

1. Set $k=0$.
2. Compute $\mathbf{p}^{(k)}$ from $\mathbf{p}^{(k)} = -\mathbf{g}(\mathbf{x}^{(k)})$. *steepest descent*
3. Compute $\alpha^{(k)}$ such that $f(\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}) = \min_{\alpha} (f(\mathbf{x}^{(k)} + \alpha\mathbf{p}^{(k)}))$ by a line search
4. Compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$.
5. If $\mathbf{x}^{(k+1)}$ satisfies given convergence criteria, then stop.
6. Set $k=k+1$, go to 2.

The proof of the convergence of the sequence $\{\mathbf{x}^{(k)}\}$ to a minimizer of f is the same as that of the general descent method, assuming that f has continuous second partials [36].

One advantage of using this steepest descent algorithm is that the sequence $\{\mathbf{x}^{(k)}\}$ generated by *Algorithm 2.3.1* will converge for any given initial estimate $\mathbf{x}^{(0)}$ in the domain of f . This enables it to be regarded as a general purpose minimization method. However, the rate of convergence of *Algorithm 2.3.1* is only linear [26], which is considered slow. The computations involved in *Algorithm 2.3.1* are simple except for the computations required for performing a line search which may vary depending on the search method used.

2.4 Newton's Method

To derive Newton's method for minimization problems, we need a stronger assumption than that of section 2.3 to start with. In this section, function $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ will be assumed to have continuous third partial derivatives in a convex neighborhood D of a critical point $\mathbf{x}^* \in \mathfrak{R}^n$ and its Hessian matrix $\mathbf{H}(\mathbf{x}^*)$ being positive definite. Then, by *Theorem 2.1.2*, \mathbf{x}^* is a local minimizer of f . This yields the necessary condition

$$(2.4.1) \quad \mathbf{g}(\mathbf{x}^*)=0,$$

where $\mathbf{g}(\mathbf{x})=(g_1(\mathbf{x}), \dots, g_n(\mathbf{x}))=(\partial_1 f(\mathbf{x}), \dots, \partial_n f(\mathbf{x}))$ is the gradient of f .

If we could solve (2.4.1), then the solution would be a minimizer of f . However, in general, (2.4.1) is a system of non-linear equations. To solve it, iterative methods have to be used.

Consider $g_i(\mathbf{x})=\partial_i(f(\mathbf{x}))$ over D . Now, each $g_i(\mathbf{x})$ has continuous second order partials by hypothesis. Hence, for an estimate $\hat{\mathbf{x}} \in D$, we have, for each $\mathbf{x} \in D$, the Taylor expansion of f around $\hat{\mathbf{x}}$,

$$(2.4.2) \quad g_i(\mathbf{x}) \approx g_i(\hat{\mathbf{x}}) + \sum_{j=1}^n \partial_j g_i(\hat{\mathbf{x}})(x_j - \hat{x}_j) + \frac{1}{2} \sum_{j=1}^n \sum_{l=1}^n \partial_j \partial_l g_i(\mathbf{y}_i)(x_j - \hat{x}_j)(x_l - \hat{x}_l),$$

where, $\mathbf{y}_i = \hat{\mathbf{x}} + \theta_i(\mathbf{x} - \hat{\mathbf{x}})$, for some $\theta_i \in (0,1)$, all $i=1, \dots, n$.

If \mathbf{x} is sufficiently close to $\hat{\mathbf{x}}$, the last term in (2.4.2) could be dropped. Then, letting $\mathbf{x}=\hat{\mathbf{x}}$ and taking into account of (2.4.1), we would obtain

$$(2.4.3) \quad \sum_{j=1}^n \partial_j g_i(\hat{\mathbf{x}})(x_j^* - \hat{x}_j) \cong -g_i(\hat{\mathbf{x}}).$$

Since the approximation (2.4.3) are true for all $i=1, \dots, n$, we then have, using the Hessian matrix of f at $\hat{\mathbf{x}}$,

$$(2.4.4) \quad \mathbf{H}(\hat{\mathbf{x}})(\mathbf{x}^* - \hat{\mathbf{x}}) \cong -\mathbf{g}(\hat{\mathbf{x}}).$$

Hence

$$(2.4.5) \quad \mathbf{x}^* \cong \hat{\mathbf{x}} - \mathbf{H}(\hat{\mathbf{x}})^{-1} \mathbf{g}(\hat{\mathbf{x}}).$$

(2.4.5) suggests that, for an initial estimate $\mathbf{x}^{(0)}$ of a solution \mathbf{x}^* of (2.4.1), we can approximate \mathbf{x}^* with arbitrary accuracy by generating the sequence $\{\mathbf{x}^{(k)}\}$ via the iteration

$$(2.4.6) \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{H}(\mathbf{x}^{(k)})^{-1} \mathbf{g}(\mathbf{x}^{(k)}), k=1, 2, \dots$$

This iterative procedure for estimating a critical point of f is called Newton's method.

Unlike the derivation of the Steepest Descent method which involves using a first order Taylor approximation of f , the derivation of the Newton method requires the uses of second order Taylor representation of f , which approximates f much better than does the first order one. This yields a much better rate of convergence for Newton's method to minimize a function than that of the Steepest Descent method, though more conditions have to be met in order for Newton's method to converge.

Note that Newton's method could be regarded as a special case of the general descent method of section 2.2 by taking the descent direction

$$(2.4.7) \quad \mathbf{p}^{(k)} = -\mathbf{H}(\mathbf{x}^{(k)})^{-1} \mathbf{g}(\mathbf{x}^{(k)})$$

at each iteration. However, the original Newton method does not include the second fundamental step for determining a step size of movement at each iteration, as we have mentioned in section 2.1. This can cause some unstable behavior of the Newton method. Therefore, to safeguard Newton's method against failures caused by lack of validity of

approximating f by the second order Taylor representation, the fundamental step for choosing $\alpha^{[k]}$ has been incorporated into the original Newton method to ensure the validity of the approximation of f . The following algorithm is based on this modified Newton's method.

Algorithm 2.4.1: Let an estimate $\mathbf{x}^{(0)}$ of an unconstrained minimizer \mathbf{x}^* of f be given.

1. Set $k=0$.
2. Compute $\mathbf{g}^{(k)}$ and $\mathbf{H}^{(k)}$ via $\mathbf{g}_i^{(k)} = \partial_i f(\mathbf{x}^{(k)})$ and $H_{ij}^{(k)} = \partial_j \partial_i f(\mathbf{x}^{(k)})$, $i, j=1, \dots, n$.
3. Compute $\mathbf{p}^{(k)}$ by solving the system of linear equations

$$\mathbf{H}^{(k)} \mathbf{p}^{(k)} = -\mathbf{g}^{(k)}.$$
4. Compute $\alpha^{(k+1)}$ using one of the line search methods mentioned in 2.1.
5. Compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$.
6. If $\mathbf{x}^{(k+1)}$ satisfies given convergence criteria, then stop.
7. Set $k=k+1$, go to 2.

The sequence $\{\mathbf{x}^{(k)}\}$ generated by *Algorithm 2.4.1* will converge to a minimizer of f if some other conditions are met [36] and it can be shown that the rate of convergence of *Algorithm 2.4.1* (without step 4 in *Algorithm 2.4.1*) is of order 2, if it does converge. This is the fastest rate normally encountered in non-linear optimization [28].

However, Newton's method is still subject to the following causes of failure during the $(k+1)$ -st iteration.

1. The direction $\mathbf{p}^{(k)}$ is orthogonal to $\mathbf{g}^{(k)}$ or nearly orthogonal to $\mathbf{g}^{(k)}$.
2. $\mathbf{H}^{(k)-1}$ exists but is not positive definite.
3. $\mathbf{H}^{(k)-1}$ does not exist.

To overcome these deficiencies of Newton's method, other techniques have to be incorporated into *Algorithm 2.4.1* [36]. Note that the case of Failure 1 could be detected

by using the condition $\left| \mathbf{g}^{(k)T} \mathbf{p}^{(k)} \right| \leq \epsilon \|\mathbf{g}^{(k)}\|_2 \|\mathbf{p}^{(k)}\|_2$, for some sufficiently small number ϵ , and, if this is the case at an iteration, we then could simply take a steepest descent step, i.e., taking $\mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$. We could also safeguard Newton's method against Failure 3 by taking a steepest descent step whenever the system of linear equations for $\mathbf{p}^{(k)}$ is not solvable. For the remaining failure case, we could take $\mathbf{p}^{(k)} = -\mathbf{p}^{(k)}$ or a steepest descent step if it does happen.

As we can see in the above discussion, one basic step that we could take, when Newton's method fails to yield a downhill step, is a steepest descent step, which is always a downhill step. Hence, we could combine the Steepest Descent method and the Newton method to yield a hybrid algorithm which is more stable than Newton's method. However, we are not going any further in this line of thought.

If the function f is the non-linear least squares function, then we will have a uniform treatment for safeguarding the Newton method. Such an approach involves the ideas of Levenberg and Marquardt that we shall discuss shortly.

The next section introduces the conjugate gradient method that is derived from more profound mathematics.

2.5 Conjugate Gradient Descent Method

The following definition and facts are needed for the establishment of the conjugate gradient descent method [36].

Definition 2.5.1: Let \mathbf{A} be a symmetric $n \times n$ matrix. Then the vectors $\mathbf{p}^{(i)}$ ($i=0, 1, \dots$) are \mathbf{A} -conjugate if and only if $\mathbf{p}^{(i)T} \mathbf{A} \mathbf{p}^{(j)} = 0$ ($i \neq j$).

Theorem 2.5.1: If 1. \mathbf{A} is an $n \times n$ symmetric positive definite matrix;

2. $\mathbf{p}^{(k)} \neq 0$ ($k = 0, \dots, n-1$) are \mathbf{A} -conjugate;

3. \mathbf{v} is any vector in \mathfrak{R}^n ,

then, (a) $\mathbf{p}^{(k)}$ ($k = 0, \dots, n-1$) form a basis for \mathfrak{R}^n ;

$$(b) \mathbf{v} = \sum_{j=0}^{n-1} \frac{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{v}}{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{p}^{(j)}} \mathbf{p}^{(j)}. \text{ (see [36] for a proof)}$$

We are interested in a quadratic function $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ of the form

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c,$$

where \mathbf{A} is an $n \times n$ symmetric positive definite matrix, \mathbf{b} is an $n \times 1$ vector, and c is a real number. Obviously, such a quadratic function has a global minimizer, where the gradient vector \mathbf{g} of f vanishes. That is, taking the derivative of f with respect to \mathbf{x} , we obtain

$$\begin{aligned} \mathbf{g}(\mathbf{x}) &= \nabla f(\mathbf{x}) = \nabla \left(\frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} \right) + \nabla (\mathbf{b}^T \mathbf{x}) + \nabla (c) \\ &= \frac{1}{2} (\mathbf{A} + \mathbf{A}^T) \mathbf{x} + \mathbf{b}, \quad i.e., \end{aligned}$$

$$(2.5.1) \quad \mathbf{g}(\mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{b} \quad (\text{since } \mathbf{A} \text{ is symmetric}).$$

If \mathbf{x}^* is a minimizer of f , then

$$(2.5.2) \quad \mathbf{g}(\mathbf{x}^*) = \mathbf{A} \mathbf{x}^* + \mathbf{b} = 0 \quad \text{or equivalently}$$

$$(2.5.3) \quad \mathbf{x}^* = -\mathbf{A}^{-1} \mathbf{b}, \quad \text{since } \mathbf{A} \text{ is positive definite and hence non-singular.}$$

Now, let us apply the general descent algorithm to the above quadratic function.

Suppose we choose the descent directions $\mathbf{p}^{(k)}$, $k=0, \dots, n-1$, such that

- (Assumption:) 1. $\mathbf{g}(\mathbf{x}^{(0)})^T \mathbf{p}^{(0)} < 0$, where $\mathbf{x}^{(0)}$ is an initial estimate
 2. $\mathbf{p}^{(k+1)}$ chosen such that $\mathbf{p}^{(k+1)T} \mathbf{A} \mathbf{p}^{(j)} = 0$, $(j=0, \dots, k)$

This gives us

$$(2.5.4) \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \sum_{j=0}^k \alpha^{(j)} \mathbf{p}^{(j)}, \quad k=0, \dots, n-1.$$

Plugging (2.5.4) into (2.5.1), we get

$$(2.5.5) \quad \begin{aligned} \mathbf{g}^{(k+1)} &= \mathbf{g}(\mathbf{x}^{(k+1)}) = \mathbf{A} \mathbf{x}^{(k+1)} + \mathbf{b} \\ &= \mathbf{A} \mathbf{x}^{(0)} + \mathbf{b} + \sum_{j=0}^k \alpha^{(j)} \mathbf{A} \mathbf{p}^{(j)}. \end{aligned}$$

Since by (2.2.5), $\mathbf{g}^{(k+1)T} \mathbf{p}^{(k)} = 0$, multiplying $\mathbf{p}^{(k)T}$ to both side of (2.5.5), and then solving for $\alpha^{(k)}$, by hypothesis 2 above, the resulting solution will simplify to

$$(2.5.6) \quad \alpha^{(k)} = -\frac{\mathbf{p}^{(k)T} \mathbf{A} \mathbf{x}^{(0)}}{\mathbf{p}^{(k)T} \mathbf{A} \mathbf{p}^{(k)}} - \frac{\mathbf{p}^{(k)T} \mathbf{b}}{\mathbf{p}^{(k)T} \mathbf{A} \mathbf{p}^{(k)}}, \quad k=0, \dots, n-1.$$

Now, taking $k=n-1$ in (2.5.4), we obtain

$$\begin{aligned} \mathbf{x}^{(n)} &= \mathbf{x}^{(0)} - \sum_{j=0}^{n-1} \frac{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{x}^{(0)}}{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{p}^{(j)}} \mathbf{p}^{(j)} - \sum_{j=0}^{n-1} \frac{\mathbf{p}^{(j)T} \mathbf{b}}{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{p}^{(j)}} \mathbf{p}^{(j)} \\ &= \mathbf{x}^{(0)} - \sum_{j=0}^{n-1} \frac{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{x}^{(0)}}{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{p}^{(j)}} \mathbf{p}^{(j)} - \sum_{j=0}^{n-1} \frac{\mathbf{p}^{(j)T} \mathbf{A} (\mathbf{A}^{-1} \mathbf{b})}{\mathbf{p}^{(j)T} \mathbf{A} \mathbf{p}^{(j)}} \mathbf{p}^{(j)} \\ &= \mathbf{x}^{(0)} - \mathbf{x}^{(0)} + \mathbf{A}^{-1} \mathbf{b} \quad (\text{Theorem 2.5.1}) \\ &= \mathbf{A}^{-1} \mathbf{b}. \end{aligned}$$

By (2.5.3), we have $\mathbf{x}^{(n)} = \mathbf{x}^*$, i.e., $\mathbf{x}^{(n)}$ is the minimizer of f . Hence, the minimizer \mathbf{x}^* of f can be obtained in n iterations (could be less than n iterations, see [36]).

To generalize this remarkable result to non-quadratic functions, we first note that, again by Taylor's theorem, a function $f: \mathfrak{R}^n \rightarrow \mathfrak{R}$ which has continuous second partial

derivatives at a point $\mathbf{x}^* \in \mathfrak{R}^n$ could be well approximated by the truncated second order Taylor representation, i.e.,

$$(2.5.7) \quad f(\mathbf{x}) \cong f(\mathbf{x}^*) + \mathbf{g}(\mathbf{x}^*)^T (\mathbf{x} - \mathbf{x}^*) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^*)^T \mathbf{H}(\mathbf{x}^*) (\mathbf{x} - \mathbf{x}^*).$$

Assuming \mathbf{x}^* is a critical point of f , we then have $\mathbf{g}(\mathbf{x}^*)=0$. Hence, (2.5.7) becomes

$$(2.5.8) \quad \begin{aligned} f(\mathbf{x}) &\cong f(\mathbf{x}^*) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^*)^T \mathbf{H}(\mathbf{x}^*) (\mathbf{x} - \mathbf{x}^*) \\ &= \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c, \end{aligned}$$

where, $\mathbf{A}=\mathbf{H}(\mathbf{x}^*)$, $\mathbf{b}=-\mathbf{H}(\mathbf{x}^*) \mathbf{x}^*$, and $c = f(\mathbf{x}^*) + \frac{1}{2} \mathbf{x}^{*T} \mathbf{H}(\mathbf{x}^*) \mathbf{x}^*$.

This shows that f can be approximated by a quadratic function and, hence, the above results about quadratic functions could be used. Upon the assumption that the Hessian matrix $\mathbf{H}(\mathbf{x}^*)$ is positive definite, the above quadratic function (2.5.8) can be used to find a minimizer of the original function f , which is readily seen to be the same as the minimizer of the quadratic function (2.5.8). Such consideration yields the iterative scheme for the conjugate direction algorithm, which involves the successive approximation of f by a quadratic function at each iteration. Note that the convergence property of the quadratic function does not hold for non-quadratic functions, i.e., a satisfactory minimizer will not generally be located within n iterations. For non-quadratic functions, a new sequence of $\mathbf{p}^{(k)}$'s, ($k=0, \dots, n-1$) should be constructed if the current approximation by one quadratic function is not sufficiently good. Hence, a reset of the parameters after every n iterations may seem to be a reasonable strategy to be used in algorithms for minimizing non-quadratic functions [18].

Notice that there are many methods to compute $\mathbf{p}^{(k)}$'s such that *assumption 2* above could be satisfied. One way to construct $\mathbf{p}^{(k)}$ at each iteration is to take $\mathbf{p}^{(k)}$ as a linear combination of $-\mathbf{g}^{(k)}$ and $\mathbf{p}^{(k-1)}$ and hence the name *conjugate gradient method*. Such a construction is encoded in the following conjugate gradient algorithm.

Algorithm 2.5.1: Let an estimate $\mathbf{x}^{(1)}$ of an unconstrained minimizer \mathbf{x}^* of f be given which is sufficiently close to \mathbf{x}^* .

1. Set $k=1$, $\mathbf{p}^{(1)} = \mathbf{r}^{(1)} = -\mathbf{g}(\mathbf{x}^{(1)})$.
2. Compute the Hessian matrix $\mathbf{H}^{(k)} = \mathbf{H}(\mathbf{x}^{(k)})$.
3. Compute $\alpha^{(k)}$ by using a line search technique [36] or by using the following formula $\alpha^{(k)} = \frac{\mathbf{p}^{(k)T} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)T} \mathbf{H}^{(k)} \mathbf{p}^{(k)}}$ (or using an approximation of $\mathbf{H}^{(k)}$)

[24].

4. Compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$ and $\mathbf{r}^{(k+1)} = -\mathbf{g}(\mathbf{x}^{(k+1)})$.

5. If $k=0 \bmod n$, go to 10.

6. Compute $\beta^{(k)}$ using one of the following:

- (Hestenes and Stiefel) $\beta^{(k)} = \frac{\mathbf{r}^{(k+1)T} (\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)})}{\mathbf{p}^{(k)T} (\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)})}$

- (Fletcher and Reeves) $\beta^{(k)} = \frac{\mathbf{r}^{(k+1)T} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)T} \mathbf{r}^{(k)}}$

- (Polak and Ribiere) $\beta^{(k)} = \frac{\mathbf{r}^{(k+1)T} (\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)})}{\mathbf{p}^{(k)T} \mathbf{r}^{(k)}}$

7. Compute $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k)} \mathbf{p}^{(k)}$.

8. If convergence is obtained, go to 11.

9. Set $k=k+1$, go to 2.

10. Set $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)}$ go to 2.

11. Set $\mathbf{x}_0 = \mathbf{x}^{(k+1)}$.

12. Stop.

Algorithm 2.5.1 follows the same descent pattern as we have always mentioned, i.e., first determine the search direction(Step 6, 7, given an initial $\mathbf{p}^{(1)}$) and then choose a step size (Step 3, 4). The rate of convergence of this algorithm is superlinear. If the formula given in Step 3 above is used, the algorithm suffers from the same problems as that of Newton's method, since the conditions on the Hessian matrix of f might be violated at an iteration. Hence, it might need the same treatments as we have mentioned in the discussion of the Newton method. Another treatment, which does not require the computation of the Hessian matrix $\mathbf{H}(\mathbf{x})$ --using a line search instead, can be found in [36].

2.6 Least Squares Minimization

In this section, we specialize the above methods to a particular kind of functions that is a sum of the squares of non-linear functions. The special form of this kind of functions enables us to describe the methods in more detail and, as we shall see, to derive a new method, namely, the Gauss-Newton method.

Consider a function of l real variables x_i ($i=1, \dots, l$) and n real variables w_j ($j=1, \dots, n$) $Y: \mathfrak{R}^l \times \mathfrak{R}^n \rightarrow \mathfrak{R}$ be a function . Writing them in vector form, we have

$$\mathbf{t}=(t_1, \dots, t_l)^T \in \mathfrak{R}^l \text{ and } \mathbf{w}=(w_1, \dots, w_n)^T \in \mathfrak{R}^n.$$

Furthermore, let m vectors \mathbf{x}_i ($i=1, \dots, m$) and m real numbers y_i . be given. We are interested in estimating a local minimizer of $E: \mathfrak{R}^n \rightarrow \mathfrak{R}$ defined, as a function of \mathbf{w} , by

$$(2.6.1) \quad E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^m (Y(\mathbf{x}_i, \mathbf{w}) - y_i)^2.$$

Let

$$(2.6.2) \quad \mathbf{V}(\mathbf{w}) = (v_1(\mathbf{w}), \dots, v_m(\mathbf{w}))^T,$$

where $v_i(\mathbf{w}) = Y(\mathbf{x}_i, \mathbf{w})$, $i=1, \dots, m$, and $\mathbf{y} = (y_1, \dots, y_m)^T$. Then (2.6.1) can be written as

$$(2.6.3) \quad E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^m (Y(\mathbf{x}_i, \mathbf{w}) - y_i)^2 = \frac{1}{2} \|\mathbf{V}(\mathbf{w}) - \mathbf{y}\|_2^2.$$

Let $f: \mathfrak{R}^n \rightarrow \mathfrak{R}^m$ be defined by

$$(2.6.4) \quad \begin{aligned} \mathbf{f}(\mathbf{w}) &= \mathbf{V}(\mathbf{w}) - \mathbf{y} \\ &= (f_1(\mathbf{w}), \dots, f_m(\mathbf{w}))^T = (v_1(\mathbf{w}) - y_1, \dots, v_m(\mathbf{w}) - y_m)^T. \end{aligned}$$

Then, (2.6.1) can also be written in vector form as

$$(2.6.5) \quad E(\mathbf{w}) = \frac{1}{2} \mathbf{f}(\mathbf{w})^T \mathbf{f}(\mathbf{w}).$$

The gradient vector $\mathbf{g}(\mathbf{w}) = (g_1(\mathbf{w}), \dots, g_m(\mathbf{w}))$ of $E(\mathbf{w})$ is given by

$$(2.6.6) \quad \begin{aligned} g_i(\mathbf{w}) &= \frac{\partial E}{\partial w_i} = \sum_{j=1}^m f_j(\mathbf{w}) \frac{\partial f_j(\mathbf{w})}{\partial w_i} \\ &= \sum_{j=1}^m f_j(\mathbf{w}) \partial_i f_j(\mathbf{w}), \quad i = 1, \dots, n. \end{aligned}$$

Using the Jacobian matrix $\mathbf{J}_{m \times n}$ of $\mathbf{f}(\mathbf{w})$, whose element is given by $J_{ij}(\mathbf{w}) = \partial_j f_i(\mathbf{w})$, $i=1, \dots, m$ and $j=1, \dots, n$, we have

$$(2.6.7) \quad \mathbf{g}(\mathbf{w}) = \mathbf{J}(\mathbf{w})^T \mathbf{f}(\mathbf{w}).$$

The Hessian matrix $\mathbf{H}(\mathbf{w})$ of $E(\mathbf{w})$ can be decomposed via the following treatment.

Consider the ij -element H_{ij} of $\mathbf{H}(\mathbf{w})$,

$$\begin{aligned} H_{ij}(\mathbf{w}) &= \frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j} = \frac{\partial}{\partial w_i} (g_j(\mathbf{w})) = \frac{\partial}{\partial w_i} \left(\sum_{k=1}^m f_k(\mathbf{w}) \partial_j f_k(\mathbf{w}) \right) \\ &= \sum_{k=1}^m \left(\partial_i f_k(\mathbf{w}) \partial_j f_k(\mathbf{w}) + f_k(\mathbf{w}) \partial_i \partial_j f_k(\mathbf{w}) \right), \quad i, j = 1, \dots, n. \end{aligned}$$

Now, if we let \mathbf{T}_i be the Hessian Matrix of f_i , i.e.,

$$(2.6.8) \quad \mathbf{T}_i(\mathbf{w}) = \nabla^2 f_i(\mathbf{w}) = (\partial_j \partial_k (f_i(\mathbf{w})))_{n \times n}, \quad i=1, \dots, m,$$

then a decomposition of the Hessian matrix of $E(\mathbf{w})$ is

$$(2.6.9) \quad \begin{aligned} \mathbf{H}(\mathbf{w}) &= \mathbf{J}(\mathbf{w})^T \mathbf{J}(\mathbf{w}) + \sum_{i=1}^m f_i(\mathbf{w}) \mathbf{T}_i(\mathbf{w}) \\ &= \mathbf{J}(\mathbf{w})^T \mathbf{J}(\mathbf{w}) + \mathbf{S}(\mathbf{w}), \end{aligned}$$

where, $\mathbf{S}(\mathbf{w}) = \sum_{i=1}^m f_i(\mathbf{w}) \mathbf{T}_i(\mathbf{w})$.

Clearly, the decomposition (2.6.9) of the Hessian matrix displays a considerable structure and hence gives us more alternatives when applying Newton's method to the function $E(\mathbf{w})$.

Now, let us apply all the methods introduced in the previous sections to the function $E(\mathbf{w})$. For the Steepest Descent method and the Conjugate Gradient method, all we need is the gradient of $E(\mathbf{w})$ at each iteration and hence equation (2.6.7) can be utilized to do so. For Newton's method, the gradient of $E(\mathbf{w})$ and the Hessian matrix of $E(\mathbf{w})$ are needed. Thus, equations (2.6.7), (2.6.8), and (2.6.9) can be used for the computations. One alternative when using the Newton method is that we could drop the term $\mathbf{S}(\mathbf{w})$ in (2.6.9) if $E(\mathbf{w})$ is expanded at a critical point \mathbf{x}^* for points sufficiently close to \mathbf{x}^* . Then the Hessian matrix $\mathbf{H}(\mathbf{w})$ of $E(\mathbf{w})$ is approximated by

$$(2.6.10) \quad \mathbf{H}(\mathbf{w}) \cong \mathbf{J}(\mathbf{w})^T \mathbf{J}(\mathbf{w}).$$

This changed form of the Newton method is called the Gauss-Newton method [36]. In this method, the Hessian matrix $\mathbf{H}(\mathbf{w})$ is computed approximately by using only the first order partials of $E(\mathbf{w})$, which is less costly in computations. The following is an algorithmic implementation of the Gauss-Newton modification of the Newton method.

Note that, like *Algorithm 2.4.1* of the Newton method, the fundamental step for choosing $\alpha^{[k]}$ has been incorporated into the original Gauss-Newton method, which does not have such a step, to ensure the validity of the approximation of $E(\mathbf{w})$.

Algorithm 2.6.1: Let an estimate $\mathbf{w}^{(0)}$ of an unconstrained minimizer \mathbf{w}^* of $E(\mathbf{w})$ be given.

1. Set $k=0$.
2. Compute $\mathbf{g}^{(k)}$ and $\mathbf{J}^{(k)}$ via

$$\mathbf{g}_i^{(k)} = \partial_i E(\mathbf{w}^{(k)}) \text{ and}$$

$$J_{ij}^{(k)} = \partial_{ij} f(\mathbf{w}^{(k)}), \quad i, j = 1, \dots, n.$$
3. Compute $\mathbf{p}^{(k)}$ by solving the system of linear equations

$$(\mathbf{J}^{(k)T} \mathbf{J}^{(k)}) \mathbf{p}^{(k)} = -\mathbf{g}^{(k)}.$$
4. Compute $\alpha^{(k+1)}$ by using one of the line search methods mentioned in 2.1.
5. Compute $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$.
6. If $\mathbf{w}^{(k+1)}$ satisfies given convergence criteria, then stop.
7. Set $k=k+1$, go to 2.

However, in the Gauss-Newton version of the Newton method, all the drawbacks of Newton's method still remain and may become worse since the approximated Hessian matrix is probably more vulnerable to singular problems. In the next section, we shall see that the Levenberg-Marquardt method could be very helpful to handle all those drawbacks.

2.7 The Levenberg-Marquardt Method

The formulation of the Levenberg-Marquardt method, as often seen in the optimization literature, is based on the Gauss-Newton method for dealing with non-linear least squares problems. When the Gauss-Newton method is utilized for a minimization problem, we need to solve the system of linear equations given by

$$(2.7.1) \quad (\mathbf{J}^{(k)T} \mathbf{J}^{(k)}) \mathbf{p}^{(k)} = -\mathbf{g}^{(k)} = -\mathbf{J}^{(k)} \mathbf{f}^{(k)},$$

and then to update the estimate via

$$(2.7.2) \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}.$$

As we have seen in section 2.6, depending on the matrix $\mathbf{J}^{(k)T} \mathbf{J}^{(k)}$, the Gauss-Newton method sometimes suffers the following problems in practice:

1. The direction $\mathbf{p}^{(k)}$ is orthogonal or nearly orthogonal to $\mathbf{g}^{(k)}$.
2. $(\mathbf{J}^{(k)T} \mathbf{J}^{(k)})^{-1}$ does not exist.

Failure 1 above generally results in little or no progress of the minimization process or something even worse. When case 2 happens, there is no way to update the variables and hence some alternative methods need to be used to handle the problem. As mentioned also in section 2.4, the solution to both of the problems is to take a steepest descent step if either of the cases does occur.

We have seen that one important choice at each iteration of every gradient-based method is to choose the search direction $\mathbf{p}^{(k)}$. Progress will be made if this direction is close to that of $-\mathbf{g}^{(k)}$. Levenberg [23] and Marquardt [24] have described methods for determining a direction, between $\mathbf{p}^{(k)}$ and $-\mathbf{g}^{(k)}$, that will ensure that the process makes progress even if either of the above cases occur. In their treatment, the essential idea is that the update $\Delta^{(k)} = \alpha^{(k)} \mathbf{p}^{(k)}$ at each iteration should be determined by solving one of the following system of linear equations.

$$(2.7.3) \quad (\mathbf{J}^{(k)T} \mathbf{J}^{(k)} + \lambda^{(k)} \mathbf{I}) \Delta^{(k)} = -\mathbf{g}^{(k)}, \text{ or}$$

$$(2.7.4) \quad (\mathbf{J}^{(k)T} \mathbf{J}^{(k)} + \lambda^{(k)} \mathbf{D}) \Delta^{(k)} = -\mathbf{g}^{(k)},$$

where $\lambda^{(k)} > 0$ is a real number and \mathbf{D} is a diagonal matrix whose elements are the diagonal elements of $\mathbf{J}^{(k)T} \mathbf{J}^{(k)}$.

In his treatment [23], Levenberg linearized the functions f_i 's in (2.7.5) below by their

$$(2.7.5) \quad E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^m f_i^2, \quad \text{where } f_i = Y_i(\mathbf{x}_i, \mathbf{w}) - y_i$$

first order Taylor expansions to obtain the approximation $\bar{E}(\mathbf{w})$ of $E(\mathbf{w})$ at an initial point \mathbf{w}_0 by the following.

$$(2.7.6) \quad \bar{E}(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^m \left(f_i(\mathbf{w}_0) + \left. \frac{\partial f_i}{\partial \mathbf{w}} \right|_{\mathbf{w}_0} \Delta \mathbf{w} \right)^2$$

Since (2.7.6) generally gives poor approximation when $\Delta \mathbf{w}$ is large in absolute value, Levenberg used the following minimization scheme in order to minimize the sum of squares of the residuals and to limit the step size simultaneously.

$$(2.7.7) \quad \bar{\bar{E}}(\mathbf{w}) = \frac{1}{\lambda} \bar{E}(\mathbf{w}) + \sum_{j=1}^n a_j (\Delta w_j)^2,$$

where λ^{-1} is a positive number expressing the relative importance of the residuals and increments in the minimization process and a_j 's are positive constants that represent the relative importance of damping the different increments. In his reasoning, λ^{-1} can be determined approximately by

$$(2.7.8) \quad \lambda^{-1} = \frac{2 \sum_{j=1}^n a_j \left(\sum_{i=1}^n \left. \frac{\partial f_i}{\partial w_j} \right|_{\mathbf{w}_0} \cdot f_i \right)^2}{E(\mathbf{w}_0)}.$$

The choices of the weighting constants a_j 's are arbitrary. One way is to set them all equal to unity, in which case, it can be shown [23] that the system of equations we need to solve to determine the step size at each iteration is

$$(2.7.9) \quad (\mathbf{J}^{(k)T} \mathbf{J}^{(k)} + \lambda^{(k)} \mathbf{I}) \Delta^{(k)} = -\mathbf{g}^{(k)}, \text{ where } \lambda^{(k)} \text{ is given by (2.7.8).}$$

This method is called Levenberg I. Another strategy (Levenberg II) to choose a_j 's is to let

$$(2.7.10) \quad a_j = \sum_{i=1}^n \left(\frac{\partial f_i}{\partial w_j} \Big|_{\mathbf{w}_0} \right)^2, \quad j=1, \dots, n.$$

Then, the system of equations we need to solve becomes [23]

$$(2.7.11) \quad (\mathbf{J}^{(k)T} \mathbf{J}^{(k)} + \lambda^{(k)} \mathbf{D}) \Delta^{(k)} = -\mathbf{g}^{(k)},$$

where $\lambda^{(k)}$ is given by (2.7.8) and \mathbf{D} is a diagonal matrix whose elements are the diagonal elements of $\mathbf{J}^{(k)T} \mathbf{J}^{(k)}$.

Levenberg's method (Levenberg I or Levenberg II) sometimes fail because the minimization process on the first iteration may lead to a very small λ that results in a very long step that is nearly orthogonal to $-\mathbf{g}$ to a point far from the global minimum, from which the method never returns. Marquardt [24] invented a better strategy for selecting λ . He started off with the Gauss-Newton method and came up with the system equations (2.7.4) whose solutions would determine both of the direction and size of the next step in the minimization process. In his treatment, he showed that the minimization process would make progress if $\lambda^{(k)}$ was sufficiently large. The validity of his formulation is based on the following three theorems cited from [36].

- Theorem 2.7.1:* If
1. $\lambda > 0$ is arbitrary;
 2. $\mathbf{Q} = \mathbf{J}^T \mathbf{J}$, where \mathbf{J} is an $m \times n$ matrix;
 3. $\mathbf{v} \in \mathfrak{R}^n$ and $\mathbf{v} \neq \mathbf{0}$;
 4. $(\mathbf{Q} + \lambda \mathbf{I}) \Delta_0 = -\mathbf{J}^T \mathbf{v}$;
 5. $\Omega = \{\Delta \in \mathfrak{R}^n \mid \|\Delta\|_2 = \|\Delta_0\|_2\}$;
 6. $\phi(\Delta) = \|\mathbf{J}\Delta + \mathbf{v}\|_2^2$

then $\phi(\Delta_0) = \min_{\Delta \in \Omega} \{\phi(\Delta)\}$.

- Theorem 2.7.2:* If
1. $\lambda > 0$ is arbitrary;
 2. $\mathbf{Q} = \mathbf{J}^T \mathbf{J}$, where \mathbf{J} is an $m \times n$ matrix;
 3. $\mathbf{v} \in \mathfrak{R}^n$ and $\mathbf{v} \neq \mathbf{0}$;
 4. $(\mathbf{Q} + \lambda \mathbf{I}) \Delta(\lambda) = -\mathbf{J}^T \mathbf{v}$

then $\|\Delta(\lambda)\|_2^2$ is a continuous monotone decreasing function of λ and $\|\Delta(\lambda)\|_2^2 \rightarrow 0$, as $\lambda \rightarrow 0$.

- Theorem 2.7.3:* If
1. $\lambda > 0$ is arbitrary;
 2. $\mathbf{Q} = \mathbf{J}^T \mathbf{J}$, where \mathbf{J} is an $m \times n$ matrix;
 3. $\mathbf{v} \in \mathfrak{R}^n$ and $\mathbf{v} \neq \mathbf{0}$;
 4. $(\mathbf{Q} + \lambda \mathbf{I}) \Delta = -\mathbf{J}^T \mathbf{v}$
 5. $\psi(\lambda) = \frac{-\Delta^T \mathbf{J}^T \mathbf{v}}{\|\Delta\|_2 \|\mathbf{J}^T \mathbf{v}\|_2}$

then ψ is monotone increasing function of λ and $\psi(\lambda) \rightarrow 1$, as $\lambda \rightarrow \infty$.

Proofs of the above theorems can be found in [36].

Theorem 2.7.1 says that $\Delta^{(k)}$ depends upon the choice of $\lambda^{(k)}$ at each iteration. By (2.6.3), the Gauss-Newton/Levenberg-Marquardt approximation of the $E(\mathbf{w})$ is, over a sufficiently small neighborhood of a current estimate $\hat{\mathbf{w}}$ of a local minimizer \mathbf{w}^* of $E(\mathbf{w})$,

$$(2.7.12) \quad \begin{aligned} E(\hat{\mathbf{w}} + \Delta) &= \|\mathbf{V}(\hat{\mathbf{w}} + \Delta) - \mathbf{y}\|_2^2 \\ &\equiv E(\hat{\mathbf{w}}) + 2f(\hat{\mathbf{w}})^T \mathbf{J}(\hat{\mathbf{w}})\Delta + \Delta^T (\mathbf{J}(\hat{\mathbf{w}})^T \mathbf{J}(\hat{\mathbf{w}}) + \lambda \mathbf{I}) \Delta \end{aligned}$$

next part?

Now we can see that, by *Theorem 2.7.2*, if λ is sufficiently large, then, $\|\hat{\Delta}\|_2$, where $\hat{\Delta}$ is the solution of (2.7.3) (i.e., the Levenberg-Marquardt update of $\mathbf{w}^{(k)}$), is sufficiently small so that the quadratic approximation (2.7.12) becomes valid. This ensures that $E(\hat{\mathbf{w}} + \hat{\Delta}) < E(\hat{\mathbf{w}})$, that is, the reduction of function values of $E(\mathbf{w})$ at each iteration. Notice also that, by *Theorem 2.7.3*, as λ increases, the direction of $\hat{\Delta}$ in (2.7.3) approaches that of $-\mathbf{g}(\hat{\mathbf{w}}) = f(\hat{\mathbf{w}})^T \mathbf{J}(\hat{\mathbf{w}})$. Hence, even if $\mathbf{J}^{(k)T} \mathbf{J}^{(k)}$ is singular or its solution does not lead to a downhill direction, the direction of the update $\hat{\Delta}$ can still be made downhill for $E(\mathbf{w})$ at $\hat{\mathbf{w}}$ by taking λ sufficiently big. Thus, we can ensure the reduction of function values at each iteration of the minimization process by properly setting each $\lambda^{(k)}$ big enough. This way, all the drawbacks of the original Gauss-Newton method are removed.

Both Levenberg and Marquardt have suggested that the matrix $\mathbf{Q} = \mathbf{J}^T \mathbf{J}$ should be scaled so that its diagonal elements become equal to unity, since the properties of the gradient methods are scale-variant [24]. It can be shown [23, 24, 28] that this is equivalent to solving

$$(2.7.13) \quad (\mathbf{J}^T \mathbf{J} + \lambda \mathbf{D}) \Delta = -\mathbf{g},$$

where, $\mathbf{D} = \text{diag}(Q_{11}, \dots, Q_{nn})$.

This treatment makes the method scale-invariant. However, theorem 2.7.3 fails because of this change (other two theorems still hold). Instead of approaching $-\frac{1}{\lambda}\mathbf{g}$ as $\lambda \rightarrow \infty$, the direction of the update Δ becomes

$$(2.7.14) \quad \Delta \rightarrow -\frac{1}{\lambda}\mathbf{D}^{-1}\mathbf{g}, \text{ as } \lambda \rightarrow \infty.$$

Nevertheless, we then have

$$(2.7.15) \quad \Delta^T \mathbf{g} \rightarrow -\frac{1}{\lambda} \mathbf{g}^T \mathbf{D}^{-1} \mathbf{g} < 0,$$

i.e., still downhill in the limit. Hence, all previous discussion for (2.7.3) still applies if the diagonal matrix \mathbf{D} is used instead of the identity matrix \mathbf{I} .

Following Marquardt's treatment, it should be noticed that setting $\lambda^{(k)}$ equal to zero leads to the Gauss-Newton method. Hence, at the beginning of each iteration, $\lambda^{(k)}$ is reduced by the factor ν , since a smaller $\lambda^{(k)}$ gives performance of the algorithm that is more close to that of the Gauss-Newton method.

Summarizing the above discussion, we are now ready to give the Levenberg-Marquardt version of the Gauss-Newton method.

Algorithm 2.7.1: Let an estimate $\mathbf{w}^{(0)}$ of an unconstrained minimizer \mathbf{w}^* of $E(\mathbf{w})$ be given.

1. Set $k=0$, $\lambda^{(k)}=0.01$, $\nu=10$.

2. Compute $\mathbf{g}^{(k)}$ and $\mathbf{J}^{(k)}$ via

$$g_i^{(k)} = \partial_i E(\mathbf{w}^{(k)}) \text{ and}$$

$$J_{ij}^{(k)} = \partial_j f_i(\mathbf{w}^{(k)}), \quad i, j = 1, \dots, n.$$

3. Set $\lambda^{(k)} = \frac{\lambda^{(k)}}{\nu}$.

4. Compute $\mathbf{p}^{(k)}$ by solving the system of linear equations

$$(\mathbf{J}^{(k)\top} \mathbf{J}^{(k)} + \lambda^{(k)} \mathbf{D}) \mathbf{p}^{(k)} = -\mathbf{g}^{(k)}.$$

5. Set $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{p}^{(k)}$.
6. If $E(\mathbf{w}^{(k+1)}) \geq E(\mathbf{w}^{(k)})$, then set $\lambda^{(k)} = \lambda^{(k)} \nu$ and go to 4.
7. If $\mathbf{w}^{(k+1)}$ satisfies given convergence criteria, then stop.
8. Set $k=k+1$, go to 2.

Note that there is no fundamental step 2, i.e., determining a step size, in *Algorithm 2.7.1*. Such a step can be observed in the process of increasing the value of $\lambda^{(k)}$ in order to result in the function-value reduction of $E(\mathbf{w})$. In the algorithm, as $\lambda^{(k)}$ increases, not only the direction of $\mathbf{p}^{(k)}$ is approaching a downhill direction but also the step size that is decreased by the factor of $(\lambda^{(k)})$ along that direction (see (2.7.14)). And, in the limit, the step size becomes infinitesimal along a downhill direction.

We have seen in this section that the Levenberg-Marquardt method is a very effective technique in overcoming the deficiencies of the Gauss-Newton method. The main task of this paper is to incorporate such a technique into the Newton method and hence implement this new method as an ANN learning law for training feedforward neural networks. Before doing this, we first explore some of the classical ANN learning algorithms that are based on the optimization models we have developed in this Chapter.

Having developed some of the mathematical models that we will need in the ANN studies, we next concentrate on the implementations of those models using the ANN structures. The next chapter gives full treatment of some of the classical gradient-based ANN learning laws for training fully-connected, feedforward neural networks.

3. ANN LEARNING ALGORITHMS

In this chapter, we describe some of the classical learning algorithms for training fully-connected, feedforward neural networks in detail. This includes their basic concepts, learning criteria, propagated computations, and learning algorithms.

3.1 Architecture of Feedforward ANN

In chapter one, we have seen an example of a feedforward neural network with two layers. In general, such a neural network can have any number of layers. The layers of a neural network usually have a left-to-right layout with the last one on the right being the output layer. The number of layers is defined to be the number of layers with weighted connections. Note that we treat the input layer of the network as just some connection nodes. The hidden layers of a network are all of the layers except the output layer of the network and, hence, the number of hidden layers is the number of layers in a network minus one. In theory, a feedforward neural network with at most two hidden layers can approximate any function practically encountered. However, no construction method has yet been found as how to build a three-layer neural network for every given such function[10].

Notation

The notations we will use in this paper are shown in Figure 3.1.1. In the layout, all neurons in a layer are consecutively indexed, beginning at 1, in an up-down fashion. The layers are indexed in a left-to-right order and they are identified by square-bracketed

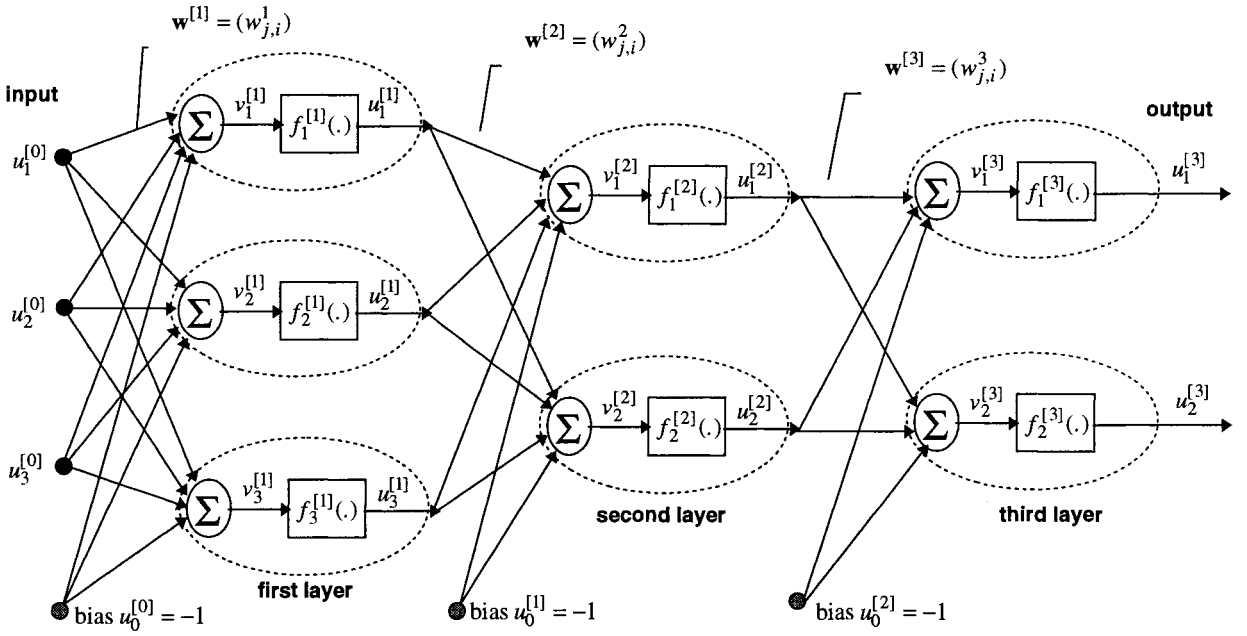


Figure 3.1.1: Architecture of a three-layered neural network

superscripts. All inputs to a neuron in layer k are denoted as $u_i^{[k-1]}$ where $i=0, 1, 2, \dots, n_{k-1}$ (n_{k-1} = number of neurons in $(k-1)$ -st layer), and in the case of $k-1=0$, then the $u_i^{[0]}$ are the inputs of the network. Here, we have assumed an extra bias node for each layer, which connects forward to each neuron in the next layer. Such nodes have no input connections from the previous layer and have a constant output value of -1, i.e., $u_0^{[k]} = -1$, for all $k=0, 1, \dots, K-1$. The weight on each of those constant connections corresponds to the bias of the neuron to which the connection is linked. Note that for each $k > 2$, $u_i^{[k-1]}$ is also the output of neuron i in the $(k-1)$ -st layer. The outputs of the network are $(u_i^{[K]})^T$, written in vector form with K being the number of layers of the network. A weight is marked as $w_{j,i}^{[k]}$, $j \neq 0$, where k is the layer index and “ j, i ” means that the weight is on the connection from the i -th neuron in layer $k-1$ to the j -th neuron in the k -th layer. In vector form, these are denoted

as $\mathbf{w}^{[k]} = (w_{j,i}^{[k]})^T$ for the weights in the k -th layer and $\mathbf{u}^{[k]} = (u_i^{[k]})^T$ for the outputs of the k -th layer and the inputs of the $(k+1)$ -st layer. The weighted sum of the inputs of a neuron, say neuron j , in layer k is denoted by

$$(3.1.1) \quad v_j^{[k]} = \sum_{i=0}^{n_k} w_{j,i}^{[k]} \cdot u_i^{[k-1]}, j \neq 0, \text{ and } v_0^{[k]} = -1.$$

Hence, the output of the neuron j in layer k can be written as

$$(3.1.2) \quad u_j^{[k]} = f_j^{[k]}(v_j^{[k]}), j \neq 0, \text{ and } u_0^{[k]} = -1, \text{ where,}$$

$f_j^{[k]}$ is the activation function of that neuron. In vector form, it will be

$$(3.1.3) \quad \mathbf{v}^{[k]} = \mathbf{w}^{[k]T} \mathbf{u}^{[k-1]} \text{ and}$$

$$(3.1.4) \quad \mathbf{u}^{[k]} = \mathbf{f}^{[k]}(\mathbf{v}^{[k]}),$$

where $\mathbf{f}^{[k]} = (f_i^{[k]})^T$ is in vector form.

The Activation function and its bias input

Perceptrons form a subclass of feedforward neural networks. In a perceptron, the activation function is a step function. This limits the applications of the perceptron networks to only classification problems. In order to introduce non-linearity into a neural network, non-linear activation functions have to be used. It is only the use of non-linear activation functions that enables multilayer neural networks to solve all kinds of mapping-approximation problems [37]. Many non-linear function will do the job, although which one to use depends upon the requirement of the learning algorithm being used. For gradient-based learning algorithms, the activation functions are required to be differentiable. The most common choices for feedforward networks are the sigmoidal

functions. Two forms of such sigmoidal functions are given below with their graphs shown in the Figures that follow [2]. Since we can scale the input and output values to within the interval (0,1) or (-1,1), there is no fundamental difference between the two except for computational considerations. In this paper, the Sigmoid function is used.

$$(3.1.5) \quad f(x) = \frac{1}{1 + e^x} \quad (\text{Sigmoid function})$$

$$(3.1.6) \quad f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (\text{Hyperbolic tangent function})$$

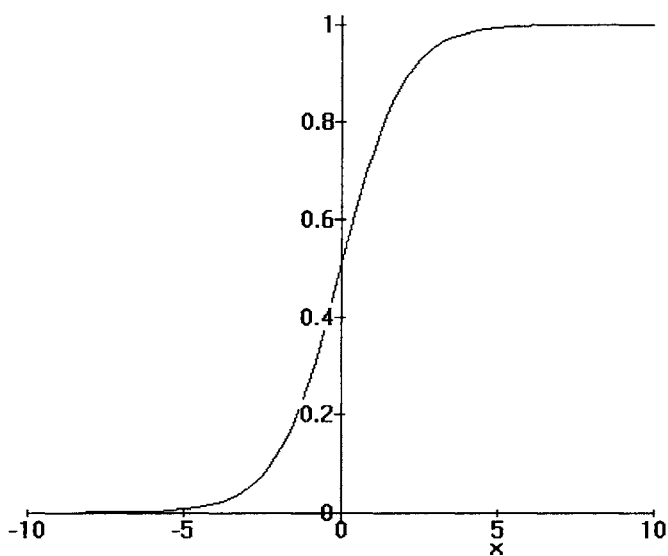
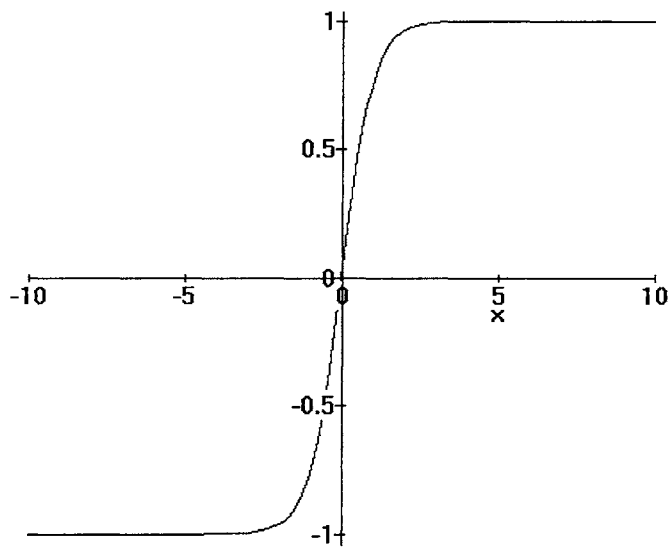


Figure 3.1.2: The Sigmoid function.



By (3.1.1) and (3.1.2) above, each neuron in a neural network defines a hyperplane in a space of dimension n_i , which is the number of variable inputs to that unit [37]. The position of this hyperplane is determined by the weights. Without a bias, the hyperplane is constrained to pass through the origin of the hyperspace. This yields some limitations on the problems that an unbiased ANN can solve. For example, without a bias, a neural network can not even solve the XOR problems without changing the domains of the input values.

Initiation of weights and bias

The weights in a neural network are initially chosen to be small random numbers. Since the activation function is usually active over a small interval and levels out outside of the interval, the slopes over the rest of the interval are very small. If the initial weights are too large, the activation functions may saturate at the beginning and the network

might get stuck in a very flat plateau or a local minimum near the starting point [16]. Thus, some optimal way to set up the initial random weights is desired. In this paper, the initial weights of all test neural networks are chosen as random numbers uniformly distributed between $\frac{-0.5}{\text{fan-in of that unit}}$ and $\frac{0.5}{\text{fan-in of that unit}}$ [10], where the fan-in of a node is the number of inputs including the bias input to that unit.

Computation in feedforward ANNs

As we have seen in Chapter 1, a neural network performs computation tasks on a layer basis. That is, when all the inputs of network are ready, the neurons in the first layer are activated and pass the results to the next layer which is in turn waiting for all its inputs to be provided. This pattern continues until the outputs of the network have all been produced. The following procedure gives an outline of such a forward pass of computations in the network, called a *forward computation* of the network.

1. The weight vectors $\mathbf{w}^{[k]}$ and the activation functions $f^{[k]}$, $k=1, 2, \dots, K$, are all given, where K is the number of layers in the network.
2. $\mathbf{u}^{[0]} = \mathbf{x}$, where \mathbf{x} is any given input vector.
3. For $k=1, 2, \dots, K$, compute $\mathbf{u}^{[k]} = f^{[k]}(\mathbf{v}^{[k]})$ by using (3.1.1) and (3.1.2).
4. $\mathbf{y} = \mathbf{u}^{[K]}$ will be the output of the network.

Mathematically, we think of a neural network as a mapping $N: \mathfrak{R}^n \rightarrow \mathfrak{R}^m$ written as $\mathbf{y} = N(\mathbf{x})$, where n and m are the dimensions of the input vector and output vector respectively. In this treatment, the layered structure of an ANN can be seen as comprising the recursive computations involved in evaluating the function N at an $\mathbf{x} \in \mathfrak{R}^n$.

3.2 Dynamic Adaptation in Feedforward ANN

A feedforward ANN exhibits dynamic changes of behavior in its training session. Based on example learning, the error made by the ANN upon an input will be realized by a pre-defined measuring function called the error function, cost function or energy function [20]. To correct the error, the error function is investigated for the sources of the error and the level of error contribution of each source, among all the neurons in the network. Then, changes of connection-weights are made in order to reduce the erroneous outputs of the network. This adaptation of behavior ends when the network has produced outputs close enough to the desired ones or when an optimal point is reached with regard to some generalization criterion. The following details the concepts involved in the above discussion of an ANN learning process.

Supervised learning

To train a feedforward ANN, supervised learning is used. This requires that sample inputs are gathered from the domain of the problem and their correct outputs provided for the quantitative realization of errors being made by the network. Normally, two such sets are chosen. One is for the training examples and the other is for the testing of the network after it has been trained. The number of examples in the training set depends on the number of weights used in the network. A general rule of thumb is that the number of samples should be larger than the number of adaptive parameters [37], preferably much larger. This rule has often been violated in practice which has led to problems of overfitting in many applications.

Another training paradigm is to divide all the samples into 3 sets [26], i.e., a learning set, a validation set, and a test set. When training an ANN, the learning process will be stopped when an optimal point is reached regarding the validation set. This way, better generalization might be achieved.

On-line and off-line learning

On-line learning means that updating of the weights takes place each time an example is presented to a neural network and errors have been produced. In off-line learning (also called batch learning), weight updating is postponed until all examples have been presented once to the network. In the on-line learning mode, the learning process is more sensitive to each individual example and, hence, it may help in escaping from local minima of the cost function [10], though no proof of such assertion has been shown yet. On the contrary, off-line learning introduces some inherent averaging and filtering effects due to the additive collection of all the weight updates. It is asserted that on-line learning is faster and more effective than the off-line learning [10], though, from the viewpoint of optimization theory, the batch learning is more consistent with the optimization models used to implement learning algorithms.

Learning criterion

In supervised training, a learning criterion is used to measure the performance of a neural network. Mathematically, this can be achieved by using an error function. Different learning algorithms implement the learning process using different kinds of

error information. For the gradient-based learning laws used in training feedforward ANNs, the error function should be continuously differentiable to the first, second, or the third order depending upon the choice of methods used. Some of the common choices are the sum-of-squared-errors function and the absolute value error function (shown below respectively). In (3.2.1), the constant $\frac{1}{2}$ is used to simplify the calculation of derivatives of $E(\mathbf{w})$ [10].

$$(3.2.1) \quad E(\mathbf{w}) = \frac{1}{2} \sum_{l=1}^s \left(\sum_{j=1}^m (f_j^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,j})^2 \right) = \frac{1}{2} \sum_{l=1}^s (\mathbf{f}^{[K]}(\mathbf{x}_l, \mathbf{w}) - \mathbf{y})^T (\mathbf{f}^{[K]}(\mathbf{x}_l, \mathbf{w}) - \mathbf{y}),$$

and

$$(3.2.2) \quad E(\mathbf{w}) = \sum_{j=1}^s \left(\sum_{i=1}^m |f_i^{[K]}(\mathbf{x}_j, \mathbf{w}) - y_{j,i}| \right),$$

where the notations are the same as those used in Chapter 1 and Chapter 2.

Learning algorithms and their evaluation

The role of the learning algorithm is to make the transition from realized errors to weight changes--thus enforcing the learning process. For the gradient-based learning algorithms, this is often achieved by employing an error function and an optimization method to minimize this error function with regard to the connection weights. The implementation of the minimization methods involves the computations of the partial derivatives of the error function. To evaluate those partial derivatives, the global error function has to be parametrized successively via the transformations made by the neurons in each layer. Such a parametrization process corresponds to a backward pass through the

neural network. Then, upon this backward parametrization, various differential items of information are obtained, and the amount of weight update is approximately computed for each of the connection weights in the network. The mathematical formulas involved in this process will be developed in section 3.3.

Different learning algorithms have different learning performances. To compare among those learning algorithms, standard measurements have been developed. The commonly used measures are the speed of learning, computation complexity, and the memory usage. For each of the learning algorithms we will introduce, we shall give, whenever possible, their performance evaluation in terms of the above three measurements.

One drawback of all the gradient-based learning algorithms that we will introduce in this paper is that the minimization process of the error function may get stuck at a local minimizer of the error function and hence stop making further progress [35]. Currently, there is no theoretical treatment to overcome this deficiency of the gradient-based learning algorithms. However, practical experiences have shown that a local minimum is not very often encountered [35]. One strategy that could be used in dealing with local minima problems is to run the learning algorithm several times, each time with a new set of initial random weights [35].

Stopping Criteria

There are several conditions that can be used to stop the learning process of an ANN.

1. The function values (or the RMS values, see below for RMS) are reduced to

within tolerance [15], i.e.,

$$E(\mathbf{w}^{(k)}) < \textit{tolerance}.$$

2. The reduction in function values at iteration $k+1$ is within tolerance [34], i.e.,

$$E(\mathbf{w}^{(k)}) - E(\mathbf{w}^{(k)} + \Delta^{(k)}) < \textit{tolerance}.$$

3. The reduction in RMS values at iteration $k+1$ is within tolerance, i.e.,

$$\text{RMS}(E(\mathbf{w}^{(k)})) - \text{RMS}(E(\mathbf{w}^{(k)} + \Delta^{(k)})) < \textit{tolerance},$$

$$\text{where, } \text{RMS}(E(\mathbf{w})) := \sqrt{\frac{E(\mathbf{w})}{\text{Number of Training Examples}}}.$$

4. The weight updates at iteration $k+1$ are within tolerance, i.e.,

$$\|\Delta \mathbf{w}^{(k)}\|_2 < \textit{tolerance}, \quad \text{or}$$

$$\|\Delta \mathbf{w}^{(k)}\|_2 < \textit{tolerance} \cdot \|\mathbf{w}^{(k)}\|_2.$$

5. (For classification problems only), each pattern has been recognized to within tolerance [35], i.e.,

$$\textit{actual output } a_j(\mathbf{w}^{(k+1)}, \mathbf{x}_l) - \textit{desired output } o_j(\mathbf{x}_l) < \textit{tolerance},$$

for all j over the output dimension and all $\mathbf{x}_l \in$ training set.

6. The number of iterations exceeds some pre-defined limit.

The stopping condition of a learning process is closely related to the generalizing ability of a neural network. To improve the generalization performance of the network, a validation set can be used while training the network and the training can be stopped once a minimum of the error on the validation set is reached [26]. This technique is often used in practice to avoid overfitting problems when training a neural network.

Generalization

The practical use of ANNs depends on their generalization capability. It is desirable that a trained ANN generalize the learning results ideally over the entire problem domain. However, this is not always practically achievable since no theoretical treatment has yet been found to control effectively the behavior of an ANN over the entire problem domain. Among the ANNs' generalization problems, overfitting is the one commonly encountered. To say that a trained ANN is overfitted on the training example means that the net conforms to the training examples, but with wild meandering outside of the training set[35], which is not desired. Such problems are usually due to the inadequate number or range of training examples. To avoid overfitting, a rule of thumb is to have the number of training examples much greater than the number of trainable weights.

3.3 Propagated Computations in Feedforward ANN

We have seen that an ANN produces its output via forward computations with the uses of the recursive formulas (3.1.3) and (3.1.4). In this section, we develop mechanics for the computations of the partial derivatives of the error function, which are needed in the implementation of the gradient-based learning algorithms.

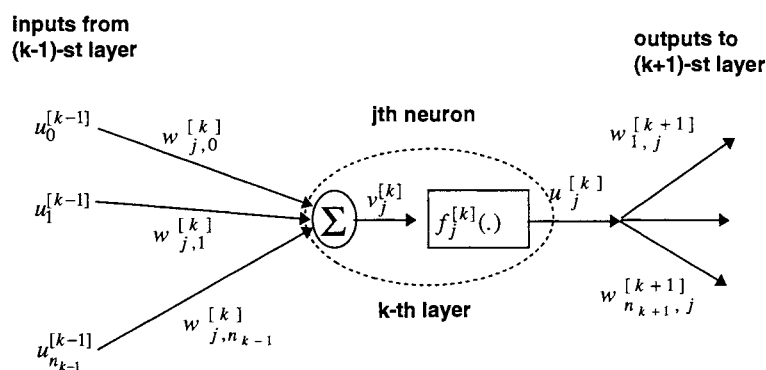


Figure 3.3.1: Part of a neural network

Throughout discussion in the rest of this chapter, we suppose a given neural network of K layers. Furthermore, we will use the sum-of-squared-errors function $E(\mathbf{w})$ defined by (3.2.1) in our discussion. Since differentiation is additive, it suffices for us to consider the following function ((3.3.1)) for one input pattern $(\mathbf{x}_l, \mathbf{y}_l)$ instead [10]. This is actually the error function used for on-line training. Summing up the derivatives over all the input samples later on would constitute the off-line training model.

$$(3.3.1) \quad E(\mathbf{w}) = \frac{1}{2} \left(\sum_{j=1}^m (f_j^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,j})^2 \right) = \frac{1}{2} (\mathbf{f}^{[K]}(\mathbf{x}_l, \mathbf{w}) - \mathbf{y}_l)^T (\mathbf{f}^{[K]}(\mathbf{x}_l, \mathbf{w}) - \mathbf{y}_l).$$

We begin by considering the network locally at a neuron, say neuron j at the k -th layer (Figure 3.3.1).

$$\text{By (3.1.2), i.e.,} \quad u_j^{[k]} = f_j^{[k]}(v_j^{[k]}), j \neq 0, \text{ and } u_0^{[k]} = -1,$$

we have

$$(3.3.2) \quad (u_j^{[k]})' = (f_j^{[k]})' := \frac{df_j^{[k]}(v_j^{[k]})}{dv_j^{[k]}} = \frac{\partial f_j^{[k]}(v_j^{[k]})}{\partial v_j^{[k]}} = (f_j^{[k]}(v_j^{[k]}))', j \neq 0,$$

and

$$(3.3.3) \quad (u_j^{[k]})'' = (f_j^{[k]})'' := \frac{d^2 f_j^{[k]}(v_j^{[k]})}{d(v_j^{[k]})^2} = \frac{\partial^2 f_j^{[k]}(v_j^{[k]})}{\partial (v_j^{[k]})^2} = (f_j^{[k]}(v_j^{[k]}))'', j \neq 0.$$

Note that the derivatives $(u_j^{[k]})'$ and $(u_j^{[k]})''$ can be calculated analytically once a proper activation function is chosen. We call such derivatives the local derivatives [8].

Using the Chain rule and (3.3.2), we have,

$$(3.3.4) \quad \frac{\partial v_j^{[k]}}{\partial w_{j,i}^{[k]}} = \frac{\partial}{\partial w_{j,i}^{[k]}} \left(\sum_{s=0}^{n_{k-1}} w_{j,s}^{[k]} u_s^{[k-1]} \right) = u_i^{[k-1]}, j \neq 0,$$

$$(3.3.5) \quad \frac{\partial u_j^{[k]}}{\partial u_i^{[k-1]}} = \frac{\partial f_j^{[k]}(v_j^{[k]})}{\partial v_j^{[k]}} \frac{\partial v_j^{[k]}}{\partial u_i^{[k-1]}} = (u_j^{[k]})' \cdot w_{j,i}^{[k]}, \quad i \neq 0,$$

$$(3.3.6) \quad \frac{\partial u_j^{[k]}}{\partial w_{j,i}^{[k]}} = \frac{\partial f_j^{[k]}(v_j^{[k]})}{\partial v_j^{[k]}} \frac{\partial v_j^{[k]}}{\partial w_{j,i}^{[k]}} = (u_j^{[k]})' \cdot u_i^{[k-1]}, \quad j \neq 0, \quad (\text{by (3.3.4)}).$$

In (3.3.4), (3.3.5), and (3.3.6), $w_{j,i}^{[k]}$ is given and $u_i^{[k-1]}$ will be known from the forward computation of the network upon a given input. Hence, the derivatives $\frac{\partial v_j^{[k]}}{\partial w_{j,i}^{[k]}}$,

$\frac{\partial u_j^{[k]}}{\partial w_{j,i}^{[k]}}$, $j \neq 0$, and $\frac{\partial u_j^{[k]}}{\partial u_i^{[k-1]}}$, $i \neq 0$, are computable for any neuron in the network once an input

is fed. Such derivatives are also called local derivatives [8]. To distinguish those local partial derivatives given by (3.3.5) and (3.3.6) from other partial derivatives that involve neurons over at least two layers, we make the following definitions [8].

$$(3.3.7) \quad (u_j^{[k]})^i := \frac{\partial u_j^{[k]}}{\partial u_i^{[k-1]}} \quad \text{and} \quad (u_0^{[k]})^i := 0, \quad i \neq 0,$$

$$(3.3.8) \quad (u_j^{[k]})^{(i)} := \frac{\partial u_j^{[k]}}{\partial w_{j,i}^{[k]}}, \quad j \neq 0,$$

In general, the computation of derivatives of the form $\frac{\partial u_j^{[k_1]}}{\partial u_i^{[k_2]}}$, where k_1 and k_2 are two

layer indices with $k_1 > k_2$, requires propagation through the network layout. We have, by the Chain Rule,

$$(3.3.9) \quad \frac{\partial u_j^{[k_1]}}{\partial u_i^{[k_2]}} = \sum_{s=1}^{n_{k_1-1}} \frac{\partial u_j^{[k_1]}}{\partial u_s^{[k_1-1]}} \frac{\partial u_s^{[k_1-1]}}{\partial u_i^{[k_2]}} = \sum_{s=1}^{n_{k_1-1}} (u_j^{[k_1]})^s \frac{\partial u_s^{[k_1-1]}}{\partial u_i^{[k_2]}}, \quad i \neq 0, \quad (\text{by 3.3.7}).$$

(3.3.9) is a forward propagation formula. The derivative of $u_j^{[k_1]}$ with regard to

$u_i^{[k_2]}$ can be recursively computed by generating the partials

$$\frac{\partial u_j^{[k]}}{\partial u_i^{[k_2]}} \text{ for } k = k_2+1, k_2+2, \dots, k_1, \quad i \neq 0,$$

with the base case at $k = k_2+1$ reducing (3.3.9) to a local derivative. Note that the sequence $k=k_2+1, k_2+2, \dots, k_1$, goes forward in the network. If $k_1 = k_2$, then the derivative is unity.

Based on (3.3.9), we can derive formula (3.3.10) for computing the partial derivatives of the form, $\frac{\partial v_j^{[k_1]}}{\partial v_i^{[k_2]}}$. By the Chain Rule, we get, for $k_1 > k_2$,

$$\begin{aligned} \frac{\partial v_j^{[k_1]}}{\partial v_i^{[k_2]}} &= \sum_{s=1}^{n_{k_1-1}} \frac{\partial v_j^{[k_1]}}{\partial u_s^{[k_1-1]}} \frac{\partial u_s^{[k_1-1]}}{\partial v_i^{[k_2]}} \\ &= \sum_{s=1}^{n_{k_1-1}} w_{j,s}^{[k_1]} \frac{\partial u_s^{[k_1-1]}}{\partial v_i^{[k_2]}} \\ &= \sum_{s=1}^{n_{k_1-1}} w_{j,s}^{[k_1]} \frac{\partial u_s^{[k_1-1]}}{\partial u_i^{[k_2]}} \frac{\partial u_i^{[k_2]}}{\partial v_i^{[k_2]}}, \text{ i.e.,} \end{aligned}$$

$$(3.3.10) \quad \frac{\partial v_j^{[k_1]}}{\partial v_i^{[k_2]}} = \sum_{s=1}^{n_{k_1-1}} w_{j,s}^{[k_1]} (u_i^{[k_2]})' \frac{\partial u_s^{[k_1-1]}}{\partial u_i^{[k_2]}}, \quad j \neq 0, \quad i \neq 0.$$

The recursion of (3.3.10) that could be seen as determined by (3.3.9) is again in a forward fashion. A special case of (3.3.10) is when $k_1 = k_2+1$, where

$$(3.3.11) \quad \begin{aligned} \frac{\partial v_j^{[k_2+1]}}{\partial v_i^{[k_2]}} &= \sum_{s=1}^{n_{k_2}} w_{j,s}^{[k_2+1]} (u_i^{[k_2]})' \frac{\partial u_s^{[k_2]}}{\partial u_i^{[k_2]}} \\ &= w_{j,i}^{[k_2+1]} (u_i^{[k_2]})', \quad j \neq 0, \quad i \neq 0. \end{aligned}$$

(since $u_s^{[k_2]}$ is not a function of $u_i^{[k_2]}$ if $s \neq i$)

Another recursive formula we shall need is the backpropagation of the partials of the

error function, i.e. $\frac{\partial E}{\partial u_j^{[k]}}$. The base of the recursion is when $k=K$. In this case, the

corresponding derivatives are defined by local derivatives and hence are computable.

That is,

$$\begin{aligned}\frac{\partial E}{\partial u_j^{[K]}} &= \frac{\partial}{\partial u_j^{[K]}} \left(\frac{1}{2} \sum_{s=1}^m (f_s^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,s})^2 \right) \\ &= \frac{1}{2} \left(\sum_{s=1}^m 2(u_s^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,s}) \frac{\partial (u_s^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,s})}{\partial u_j^{[K]}} \right)\end{aligned}$$

Since $u_s^{[K]}(\mathbf{x}_l, \mathbf{w})$ is not a function of $u_j^{[K]}$ for $s \neq j$, we have,

$$(3.3.12) \quad \frac{\partial E}{\partial u_j^{[K]}} = (u_j^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,j}) \quad , j \neq 0.$$

It is easily seen that (3.3.12) is computable after the forward computation of the network upon input \mathbf{x}_l . Note that following (3.3.12), we have

$$(3.3.13) \quad \frac{\partial^2 E}{\partial (u_j^{[K]})^2} = \frac{\partial E}{\partial u_j^{[K]}} (u_j^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,j}) = 1, \quad j \neq 0.$$

For $k < K$, we have, again by the Chain Rule,

$$(3.3.14) \quad \frac{\partial E}{\partial u_j^{[k]}} = \sum_{s=1}^{n_{k+1}} \frac{\partial E}{\partial u_s^{[k+1]}} \frac{\partial u_s^{[k+1]}}{\partial u_j^{[k]}} = \sum_{s=1}^{n_{k+1}} \frac{\partial E}{\partial u_s^{[k+1]}} (u_s^{[k+1]})^j, \quad j \neq 0.$$

To compute $\frac{\partial E}{\partial u_j^{[k]}}$, starting at the output layer, we can use (3.3.14) recursively to

generate the derivatives $\frac{\partial E}{\partial u_j^{[k]}}$ for $k=K, K-1, K-2, \dots, k_1$. The base case in the sequence is

given by (3.1.12). The recursion of (3.3.14) is readily seen to operate in a backward fashion.

When computing the Hessian matrix of $E(\mathbf{w})$, we need to calculate the term

$\frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}}$ ($j_1, j_2 \neq 0$) with $k_1 \geq k_2$. By the Chain Rule, we have [6]

$$(3.3.15) \quad \begin{aligned} \frac{\partial E}{\partial v_{j_1}^{[k_1]}} &= \sum_{s=1}^{n_{k_1+1}} \frac{\partial E}{\partial v_s^{[k_1+1]}} \frac{\partial v_s^{[k_1+1]}}{\partial v_{j_1}^{[k_1]}} \\ &= \sum_{s=1}^{n_{k_1+1}} \frac{\partial E}{\partial v_s^{[k_1+1]}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})', \quad j_1 \neq 0 \quad (\text{by (3.3.11)}) \end{aligned}$$

Consider

$$\begin{aligned} \frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}} &= \frac{\partial}{\partial v_{j_2}^{[k_2]}} \left(\frac{\partial E}{\partial v_{j_1}^{[k_1]}} \right) \\ &= \frac{\partial}{\partial v_{j_2}^{[k_2]}} \left(\sum_{s=1}^{n_{k_1+1}} \frac{\partial E}{\partial v_s^{[k_1+1]}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})' \right) \quad (\text{by (3.3.15)}) \\ &= \sum_{s=1}^{n_{k_1+1}} \frac{\partial}{\partial v_{j_2}^{[k_2]}} \left(w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})' \right) \cdot \frac{\partial E}{\partial v_s^{[k_1+1]}} + w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})' \frac{\partial}{\partial v_{j_2}^{[k_2]}} \left(\frac{\partial E}{\partial v_s^{[k_1+1]}} \right) \\ &= \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} \frac{\partial}{\partial v_{j_2}^{[k_2]}} \left((u_{j_1}^{[k_1]})' \right) \cdot \frac{\partial E}{\partial v_s^{[k_1+1]}} + \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})' \frac{\partial^2 E}{\partial v_s^{[k_1+1]} \partial v_{j_2}^{[k_2]}} \end{aligned}$$

(since $w_{s,j_1}^{[k_1+1]}$ is independent of $v_{j_2}^{[k_2]}$ for $k_1 \geq k_2$)

$$\begin{aligned} &= \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} \frac{\partial}{\partial v_{j_1}^{[k_1]}} \left((u_{j_1}^{[k_1]})' \right) \frac{\partial v_{j_1}^{[k_1]}}{\partial v_{j_2}^{[k_2]}} \cdot \frac{\partial E}{\partial v_s^{[k_1+1]}} + \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})' \frac{\partial^2 E}{\partial v_s^{[k_1+1]} \partial v_{j_2}^{[k_2]}} \\ &= \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})'' \frac{\partial v_{j_1}^{[k_1]}}{\partial v_{j_2}^{[k_2]}} \cdot \frac{\partial E}{\partial v_s^{[k_1+1]}} \frac{\partial v_s^{[k_1+1]}}{\partial v_s^{[k_1+1]}} + \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})' \frac{\partial^2 E}{\partial v_s^{[k_1+1]} \partial v_{j_2}^{[k_2]}} \end{aligned}$$

Thus, we have obtained, for $j_1, j_2 \neq 0$,

$$(3.3.16) \quad \frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}} = \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})'' \frac{\partial v_{j_1}^{[k_1]}}{\partial v_{j_2}^{[k_2]}} \cdot \frac{\partial E}{\partial v_s^{[k_1+1]}} (u_s^{[k_1+1]})' + \sum_{s=1}^{n_{k_1+1}} w_{s,j_1}^{[k_1+1]} (u_{j_1}^{[k_1]})' \frac{\partial^2 E}{\partial v_s^{[k_1+1]} \partial v_{j_2}^{[k_2]}}$$

Each factor in the right-hand side of (3.3.16) is given, local, or computable via a

previous formula, except the terms $\frac{\partial^2 E}{\partial v_s^{[k_1+1]} \partial v_{j_2}^{[k_2]}}$, whose computations involve the

recursive uses of (3.3.16). Hence, to compute $\frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}}$, we back-propagate the

computations of

$$\frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}}, \text{ for } k=K, K-1, K-2, \dots, k_1.$$

When $k=K$, we have the base case

$$\begin{aligned} \frac{\partial^2 E}{\partial v_{j_1}^{[K]} \partial v_{j_2}^{[K_2]}} &= \frac{\partial}{\partial v_{j_2}^{[K_2]}} \left(\frac{\partial E}{\partial v_{j_1}^{[K]}} \right) = \frac{\partial}{\partial v_{j_2}^{[K_2]}} \left(\frac{\partial E}{\partial u_{j_1}^{[K]}} \frac{\partial u_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} \right) = \frac{\partial}{\partial v_{j_2}^{[K_2]}} \left(\frac{\partial E}{\partial u_{j_1}^{[K]}} (u_{j_1}^{[K]})' \right) \\ &= \frac{\partial}{\partial v_{j_2}^{[K_2]}} \left(\frac{\partial E}{\partial u_{j_1}^{[K]}} \right) \cdot (u_{j_1}^{[K]})' + \frac{\partial E}{\partial u_{j_1}^{[K]}} \frac{\partial}{\partial v_{j_2}^{[K_2]}} \left((u_{j_1}^{[K]})' \right) \\ &= \frac{\partial}{\partial u_{j_1}^{[K_1]}} \left(\frac{\partial E}{\partial u_{j_1}^{[K]}} \right) \frac{\partial u_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} (u_{j_1}^{[K]})' + \frac{\partial E}{\partial u_{j_1}^{[K]}} \frac{\partial}{\partial v_{j_1}^{[K]}} \left((u_{j_1}^{[K]})' \right) \frac{\partial v_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} \\ &= \frac{\partial^2 E}{\partial (u_{j_1}^{[K]})^2} \frac{\partial u_{j_1}^{[K]}}{\partial v_{j_1}^{[K]}} \frac{\partial v_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} (u_{j_1}^{[K]})' + \frac{\partial E}{\partial u_{j_1}^{[K]}} (u_{j_1}^{[K]})'' \frac{\partial v_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} \\ &= \frac{\partial^2 E}{\partial (u_{j_1}^{[K]})^2} \frac{\partial v_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} \left((u_{j_1}^{[K]})' \right)^2 + \frac{\partial E}{\partial u_{j_1}^{[K]}} (u_{j_1}^{[K]})'' \frac{\partial v_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} \end{aligned}$$

Using (3.3.13) and factoring out a term, we then obtain [6]

$$(3.3.17) \quad \frac{\partial^2 E}{\partial v_{j_1}^{[K]} \partial v_{j_2}^{[K_2]}} = \frac{\partial v_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}} \left(\left((u_{j_1}^{[K]})' \right)^2 + \frac{\partial E}{\partial u_{j_1}^{[K]}} (u_{j_1}^{[K]})'' \right), j_1, j_2 \neq 0.$$

Since $\frac{\partial v_{j_1}^{[K]}}{\partial v_{j_2}^{[K_2]}}$ and $\frac{\partial E}{\partial u_{j_1}^{[K]}}$ can be calculated via the uses of (3.3.10) and (3.3.14), formula

(3.3.17) can be used to compute the base case of (3.3.16). Note that the calculation of

$\frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}}$ requires the forward propagation of first-order partial derivatives, the backward propagation of the first-order partials of $E(\mathbf{w})$, and the second order partials.

Those propagation formulas that we have developed in this section will enable us to derive the learning algorithms based on the methods we have introduced in Chapter 2. We carry out the derivation in the next section.

3.4 Classical Learning Algorithms

Based on the classical optimization methods introduced in Chapter 2, we develop their corresponding learning algorithms for training feedforward ANNs. The only major task that is needed to yield our learning algorithms from the algorithms of Chapter 2 is the calculation of all the first (and/or second) partial derivatives of the error function with regard to the weights. Such computations are characterized by the special recursive form of the network function given by (3.1.3) and (3.1.4). Since the error function is recursive, the computations of the derivatives are also recursive.

While the derivation of all formulas in the previous section and the current section are valid for any proper choice of an activation function, we shall use the Sigmoid function defined by (3.1.5) as the activation function for each neuron in the network. For convenience, the Sigmoid is cited below along with its derivatives [6].

$$(3.4.1) \quad f(x) = \frac{1}{1+e^{-x}} \quad (\text{Sigmoid function})$$

$$(3.4.2) \quad \begin{aligned} f'(x) &= -\frac{1}{(1+e^{-x})^2} \cdot e^{-x} = \frac{1}{(1+e^{-x})} \frac{(1+e^{-x})-1}{(1+e^{-x})} \\ &= f(x)(1-f(x)) \end{aligned}$$

$$\begin{aligned}
(3.4.3) \quad f''(x) &= (f(x)(1-f(x)))' = f'(x)(1-f(x)) - f(x)f'(x) \\
&= f(x)(1-f(x))^2 - f(x)f'(x)(1-f(x)) \\
&= f(x)(1-f(x))(1-2f(x))
\end{aligned}$$

The first-order and second-order derivatives of the Sigmoid function are needed when evaluating the local derivatives defined by (3.3.2) and (3.3.3).

Steepest Descent Learning Algorithm

For the Steepest Descent method, we need all the first order partial derivatives of the error function with regard to a connection weight. By the Chain rule, we have, for a layer index $k < K$,

$$(3.4.4) \quad \frac{\partial E}{\partial w_{j,i}^{[k]}} = \frac{\partial E}{\partial u_j^{[k]}} \frac{\partial u_j^{[k]}}{\partial w_{j,i}^{[k]}} = \frac{\partial E}{\partial u_j^{[k]}} (u_j^{[k]})^{(i)}, \quad j \neq 0.$$

Based on the recursion of $\frac{\partial E}{\partial u_j^{[k]}}$ given by (3.3.14), we can see that (3.4.4) is a backward propagation formula. The base case of the recursion (3.4.4) is when $k=K$, where we have

$$\begin{aligned}
(3.4.5) \quad \frac{\partial E}{\partial w_{j,i}^{[K]}} &= \frac{\partial E}{\partial u_j^{[K]}} (u_j^{[K]})^{(i)} \\
&= (f_j^{[K]}(\mathbf{x}_l, \mathbf{w}) - y_{l,j}) (u_j^{[K]})^{(i)}, \quad j \neq 0 \quad (\text{by (3.3.12) and (3.3.8)}).
\end{aligned}$$

The last expression in (3.4.5) is computable upon a forward computation of the network. Hence, any derivative $\frac{\partial E}{\partial w_{j,i}^{[k]}}$ of E can be computed by the uses of (3.4.4) and (3.4.5). This yields the gradient of E at \mathbf{x}_l . To determine the step size of a downhill move along the direction of the gradient, one simple *ad hoc* method is to fix a step size

beforehand. A better approach is to use a line search method as mentioned in section 2.1.

Upon the above preliminaries, we are now ready to give the first learning algorithm that is based on the Steepest Descent method.

Algorithm 3.4.1: Given a set $\mathcal{S}=\{(\mathbf{x}_l, \mathbf{y}_l) \mid \mathbf{x}_l = \text{input}, \mathbf{y}_l = \text{desired output of } \mathbf{x}_l\}$ of L training patterns and given a network setup of K layers with input dimension of n and output dimension of m .

1. Initialize all weights $w_{j,i}^{[k']}$ as random numbers uniformly distributed between

$$\frac{-0.5}{\text{fan-in of that unit}} \text{ and } \frac{0.5}{\text{fan-in of that unit}} .$$

set stopping *Tolerance*

set $\mathbf{w}^{(1)} = (w_{j,i}^{[k']})$ and $k=1$.

2. For each input $\mathbf{x}_l \in \mathcal{S}$, repeat step 3, 4, 5, and 6.

3. Compute the actual output of the network by using (3.1.3) and (3.1.4).

4. Compute the gradient $\mathbf{g}^{(k)}$ of $E(\mathbf{w})$ via the use of (3.4.4), (3.4.5), and (3.4.2), and set $\mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$.

5. Determine the step size $\alpha^{(k)}$ by using a line search technique such that

$$E(\mathbf{w}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}) = \min_{\alpha} (E(\mathbf{w}^{(k)} + \alpha \mathbf{p}^{(k)})) .$$

6. (on-line version) Update the weights $\mathbf{w}^{(k+1)} = (w_{j,i}^{[k']})^{(k+1)}$ via

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)} .$$

(off-line version) Accumulate the weight updates $\mathbf{w}^{(k+1)}$ given above over all input patterns.

7. If all the weights $\mathbf{w}^{(k+1)} = (w_{j,i}^{[k']})^{(k+1)}$ is such that the following convergence criterion is satisfied, then stop.

$$\sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k+1)}))}{L}} - \sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k)}))}{L}} < \textit{Tolerance} .$$

Otherwise, set $k=k+1$, go to 2.

The rate of convergence of *Algorithm 3.4.1* is linear. The computational complexity of this algorithm is fairly simple except for the linear search method used.

The steepest descent method is rarely used today in the field of optimization because of its slow rate of convergence. However, it is still employed in ANN learning algorithms due to its simplicity. Various modified versions for speeding up *Algorithm 3.4.1* have been proposed. However, investigation of those speeding up algorithms is out of the scope of this paper. Further development in this direction can be found in [10].

Algorithm for Computing the Hessian Matrix of E

The derivation of our proposed damped Newton learning algorithm requires the uses of both the gradient \mathbf{g} and the Hessian matrix \mathbf{H} of $E(\mathbf{w})$, which might also be needed for the Conjugate Gradient learning algorithm. The calculation of the gradient \mathbf{g} has been derived above. We now derive the formulas for computing the Hessian matrix \mathbf{H} .

The element of the Hessian matrix \mathbf{H} of $E(\mathbf{w})$ is of the form $\frac{\partial^2 E}{\partial w_{j_1, i_1}^{[k_1]} \partial w_{j_2, i_2}^{[k_2]}}$, $j_1, j_2 \neq 0$.

Since the Hessian matrix is symmetric, we only need to compute the second derivatives with $k_1 \geq k_2$. In the following discussion, without loss of generality, we shall assume $k_1 \geq k_2$. Note that this means there is no connection from neuron i_1 in layer k_1 to neuron i_2 in layer k_2 .

Considering the first order partial derivative of $E(\mathbf{w})$ locally at $w_{j, i}^{[k]}$, we have, by the Chain Rule and (3.3.4) [6],

$$(3.4.6) \quad \frac{\partial E}{\partial w_{j_1, i_1}^{[k_1]}} = \frac{\partial E}{\partial v_{j_1}^{[k_1]}} \frac{\partial v_{j_1}^{[k_1]}}{\partial w_{j_1, i_1}^{[k_1]}} = \frac{\partial E}{\partial v_{j_1}^{[k_1]}} u_{i_1}^{[k_1-1]}, j_1 \neq 0$$

Differentiating (3.4.6) with respect to $w_{j_2, i_2}^{[k_2]}$, we obtain, for $k_1 > k_2$

$$\begin{aligned} \frac{\partial^2 E}{\partial w_{j_1, i_1}^{[k_1]} \partial w_{j_2, i_2}^{[k_2]}} &= \frac{\partial}{\partial w_{j_2, i_2}^{[k_2]}} \left(\frac{\partial E}{\partial v_{j_1}^{[k_1]}} u_{i_1}^{[k_1-1]} \right) \\ &= \frac{\partial}{\partial w_{j_2, i_2}^{[k_2]}} \left(\frac{\partial E}{\partial v_{j_1}^{[k_1]}} \right) \cdot u_{i_1}^{[k_1-1]} + \left(\frac{\partial E}{\partial v_{j_1}^{[k_1]}} \right) \frac{\partial}{\partial w_{j_2, i_2}^{[k_2]}} \left(u_{i_1}^{[k_1-1]} \right) \\ &= \frac{\partial}{\partial v_{j_2}^{[k_2]}} \left(\frac{\partial E}{\partial v_{j_1}^{[k_1]}} \right) \frac{\partial v_{j_2}^{[k_2]}}{\partial w_{j_2, i_2}^{[k_2]}} \cdot u_{i_1}^{[k_1-1]} + \left(\frac{\partial E}{\partial v_{j_1}^{[k_1]}} \right) \frac{\partial u_{i_1}^{[k_1-1]}}{\partial v_{j_2}^{[k_2]}} \frac{\partial v_{j_2}^{[k_2]}}{\partial w_{j_2, i_2}^{[k_2]}} \\ &= \frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}} u_{i_2}^{[k_2-1]} u_{i_1}^{[k_1-1]} + \left(\frac{\partial E}{\partial v_{j_1}^{[k_1]}} \right) \frac{\partial u_{i_1}^{[k_1-1]}}{\partial v_{j_1}^{[k_1-1]}} \frac{\partial v_{j_1}^{[k_1-1]}}{\partial v_{j_2}^{[k_2]}} u_{i_2}^{[k_2-1]} \\ &= \frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}} u_{i_2}^{[k_2-1]} u_{i_1}^{[k_1-1]} + \left(\frac{\partial E}{\partial u_{j_1}^{[k_1]}} \frac{\partial u_{j_1}^{[k_1]}}{\partial v_{j_1}^{[k_1]}} \right) \left(u_{i_1}^{[k_1-1]} \right)' \frac{\partial v_{j_1}^{[k_1-1]}}{\partial v_{j_2}^{[k_2]}} u_{i_2}^{[k_2-1]} \end{aligned}$$

This gives us

$$(3.4.7) \quad \frac{\partial^2 E}{\partial w_{j_1, i_1}^{[k_1]} \partial w_{j_2, i_2}^{[k_2]}} = \frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}} u_{i_2}^{[k_2-1]} u_{i_1}^{[k_1-1]} + \frac{\partial E}{\partial u_{j_1}^{[k_1]}} \left(u_{j_1}^{[k_1]} \right)' \left(u_{i_1}^{[k_1-1]} \right)' \frac{\partial v_{j_1}^{[k_1-1]}}{\partial v_{j_2}^{[k_2]}} u_{i_2}^{[k_2-1]}, j_1, j_2 \neq 0.$$

When $k_1 = k_2$, since $u_{i_1}^{[k_1-1]}$ is not a function of $w_{j_2, i_2}^{[k_2]}$ in (3.4.6), we have

$$(3.4.8) \quad \frac{\partial^2 E}{\partial w_{j_1, i_1}^{[k_1]} \partial w_{j_2, i_2}^{[k_1]}} = \frac{\partial^2 E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_1]}} u_{i_1}^{[k_1-1]} u_{i_2}^{[k_1-1]}, j_1, j_2 \neq 0.$$

Each factor on the right-hand side of (3.4.7) and (3.4.8) can be computed either locally or by a propagation formula derived in the previous section. Note that the computation of

$\frac{\partial^2 E}{\partial w_{j_1, i_1}^{[k_1]} \partial w_{j_2, i_2}^{[k_2]}}$ normally requires multiple forward and backward passes through the

network, the number of which scales with the number of hidden nodes in the network [6].

The following algorithm is for the computation of the Hessian matrix of $E(\mathbf{w})$.

Algorithm 3.4.2: (Exact calculation of Hessian Matrix \mathbf{H} of $E(\mathbf{w})$)

Assume given an input pattern $(\mathbf{x}_t, \mathbf{y}_t)$

1. Compute the following

$$\{ u_j^{[k]} \mid k=1, 2, \dots, K, j=0, 1, \dots, n_k \}, \text{ by (3.1.3) and (3.1.4),}$$

$$\{ (u_j^{[k]})' \mid k=1, 2, \dots, K, j=0, 1, \dots, n_k \}, \text{ via (3.4.2),}$$

$$\{ (u_j^{[k]})'' \mid k=1, 2, \dots, K, j=0, 1, \dots, n_k \}, \text{ by (3.4.3),}$$

$$\{ (u_j^{[k]})^i \mid k=1, 2, \dots, K, j=0, 1, \dots, n_k, i=0, 1, \dots, n_{k-1} \} \text{ via (3.3.7).}$$

2. Compute $\{ \frac{\partial u_{j_1}^{[k_1]}}{\partial u_{j_2}^{[k_2]}} \mid K \geq k_1 \geq k_2 \geq 1 \}$ by forward propagation formula (3.3.9).

3. Compute $\{ \frac{\partial v_{j_1}^{[k_1]}}{\partial v_{j_2}^{[k_2]}} \mid K \geq k_1 \geq k_2 \geq 1 \}$ by forward propagation formula (3.3.10).

4. Calculate $\{ \frac{\partial E}{\partial u_j^{[k]}} \mid k=1, 2, \dots, K, j=0, 1, \dots, n_k \}$ via the back-propagation formula (3.3.14).

5. Generate $\{ \frac{\partial E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}} \mid K \geq k_1 \geq k_2 \geq 1 \}$ by using the backward recursive formula (3.3.16).

6. Evaluate $\{ \frac{\partial E}{\partial w_{j_1, i_1}^{[k_1]} \partial w_{j_2, i_2}^{[k_2]}} \mid K \geq k_1 \geq k_2 \geq 1 \}$ via (3.4.7) and (3.4.8).

7. Obtain the Hessian Matrix \mathbf{H} of $E(\mathbf{w})$ by symmetry and stop.

The Gauss Newton Learning Algorithm

The implementation of the Gauss-Newton version of Newton's method requires the calculation of the gradient of $E(\mathbf{w})$ and the Jacobian matrix \mathbf{J} of the vector function

defined by the network. The element of the Jacobian matrix is of the form

$$(3.4.9) \quad \frac{\partial u_m^{[K]}}{\partial w_{j,i}^{[k]}} = \sum_{s=1}^{n_k} \frac{\partial u_m^{[K]}}{\partial u_s^{[k]}} \frac{\partial u_s^{[k]}}{\partial w_{j,i}^{[k]}} = \frac{\partial u_m^{[K]}}{\partial u_j^{[k]}} (u_j^{[k]})' u_i^{[k-1]} \quad (\text{by (3.3.6)}).$$

By using (3.3.14), we can compute (3.4.9) via backward propagation after a forward computation through the network. The following algorithm is the Gauss-Newton learning algorithm, which is based on *Algorithm 2.6.1*.

Algorithm 3.4.3: Given a set $\mathbf{S}=\{(\mathbf{x}_l, \mathbf{y}_l) \mid \mathbf{x}_l = \text{input}, \mathbf{y}_l = \text{desired output of } \mathbf{x}_l\}$ of L training patterns and given a network setup of K layers with input dimension of n and output dimension of m .

1. Initialize all weights $w_{j,i}^{[k]}$ as random numbers uniformly distributed between

$$\frac{-0.5}{\text{fan-in of that unit}} \quad \text{and} \quad \frac{0.5}{\text{fan-in of that unit}}.$$

set stopping *Tolerance*

set $\mathbf{w}^{(1)} = (w_{j,i}^{[k]})$ and $k=1$.

2. For each pattern $(\mathbf{x}_l, \mathbf{y}_l) \in \mathbf{S}$, repeat step 2.1, 2.2, and 2.3, with $\mathbf{g}^{(k)}=0$.
 - 2.1. Using the weight $\mathbf{w}^{(k)}$, compute the actual output of the network by using (3.1.3) and (3.1.4).
 - 2.2. Obtain the gradient $\mathbf{g}(\mathbf{x}_l)$ of $E(\mathbf{w})$ via the use of (3.4.4), (3.4.5), and (3.3.8),
 - 2.3. sum up $\mathbf{g}(\mathbf{x}_l)$'s, i.e., $\mathbf{g}^{(k)} = \mathbf{g}^{(k)} + \mathbf{g}(\mathbf{x}_l)$
3. For each pattern $(\mathbf{x}_l, \mathbf{y}_l) \in \mathbf{S}$, repeat step 3.1 and 3.2 with $\mathbf{J}^{(k)}=0$.
 - 3.1. Compute the Jacobian Matrix $\mathbf{J}(\mathbf{w}^{(k)}, \mathbf{x}_l)$ via (3.4.9).
 - 3.2. sum up $\mathbf{J}(\mathbf{w}^{(k)}, \mathbf{x}_l)$'s, i.e., $\mathbf{J}^{(k)} = \mathbf{J}^{(k)} + \mathbf{J}(\mathbf{w}^{(k)}, \mathbf{x}_l)$.
4. Set $\mathbf{H}^{(k)} = \mathbf{J}^{(k)\top} \mathbf{J}^{(k)}$ and compute $\mathbf{p}^{(k)}$ by solving the system of linear equations $\mathbf{H}^{(k)} \mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$.
5. Compute $\alpha^{(k+1)}$ one of the line search methods in section 2.1.
6. Compute $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$.

7. If all the weights $\mathbf{w}^{(k+1)} = (w_{j,i}^{[k']})^{(k+1)}$ is such that the following convergence criterion is satisfied, then go to 8.

$$\sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k+1)}))}{L}} - \sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k)}))}{L}} < \textit{Tolerance} .$$

Otherwise, set $k=k+1$, go to 2.

- 8 . Set $\mathbf{w}_0 = \mathbf{w}^{(k+1)}$ and stop.

Conjugate Descent Learning Algorithm

The following learning algorithm is based on *Algorithm 2.5.1* of the conjugate descent method. It is used for the off-line training model.

Algorithm 3.4.4: Given a set $\mathcal{S} = \{(\mathbf{x}_l, \mathbf{y}_l) \mid \mathbf{x}_l = \text{input}, \mathbf{y}_l = \text{desired output of } \mathbf{x}_l\}$ of L training patterns and given a network setup of K layers with input dimension of n and output dimension of m .

1. Initialize all weights $w_{j,i}^{[k']}$ as random numbers uniformly distributed between

$$\frac{-0.5}{\text{fan-in of that unit}} \text{ and } \frac{0.5}{\text{fan-in of that unit}} .$$

Set stopping *Tolerance*

set $\mathbf{w}^{(1)} = (w_{j,i}^{[k']})$ and $k=1$.

2. For each pattern $(\mathbf{x}_l, \mathbf{y}_l) \in \mathcal{S}$, repeat step 2.1, 2.2, 2.3, 2.4, with $\mathbf{g}^{(k)} = 0$.
- 2.1. Using the weight $\mathbf{w}^{(k)}$, compute the actual output of the network by using (3.1.3) and (3.1.4).
 - 2.2. Calculate error information by using (3.3.12), based on the desired output \mathbf{y}_l ,
 - 2.3. Obtain the gradient $\mathbf{g}(\mathbf{x}_l)$ of $E(\mathbf{w})$ via the use of (3.4.4), (3.4.5), and (3.3.8),
 - 2.4. sum up $\mathbf{g}(\mathbf{x}_l)$'s, i.e., $\mathbf{g}^{(k)} = \mathbf{g}^{(k)} + \mathbf{g}(\mathbf{x}_l)$
3. If $k=1$, then set $\mathbf{p}^{(1)} = \mathbf{r}^{(1)} = -\mathbf{g}^{(1)}$.

4. Compute $\alpha^{(k)}$ by using a line search technique [36]

or by using the formula $\alpha^{(k)} = \frac{\mathbf{p}^{(k)T} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)T} \mathbf{H}^{(k)} \mathbf{p}^{(k)}}$ (or using an approximation of

$\mathbf{H}^{(k)}$) [24].

5. Compute $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$ and, using step 2 to compute $\mathbf{g}^{(k+1)}$, and set $\mathbf{r}^{(k+1)} = -\mathbf{g}^{(k+1)}$.

6. If $k=0 \bmod n$, go to 10.

7. Compute $\beta^{(k)}$ using one of the following:

- (Hestenes and Stiefel) $\beta^{(k)} = \frac{\mathbf{r}^{(k+1)T} (\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)})}{\mathbf{p}^{(k)T} (\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)})}$

- (Fletcher and Reeves) $\beta^{(k)} = \frac{\mathbf{r}^{(k+1)T} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)T} \mathbf{r}^{(k)}}$

- (Polak and Ribiere) $\beta^{(k)} = \frac{\mathbf{r}^{(k+1)T} (\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)})}{\mathbf{p}^{(k)T} \mathbf{r}^{(k)}}$

8. Compute $\mathbf{p}^{(k+1)} = -\mathbf{r}^{(k+1)} + \beta^{(k)} \mathbf{p}^{(k)}$.

9. If all the weights $\mathbf{w}^{(k+1)} = (\mathbf{w}_{j,i}^{[k+1]})^{(k+1)}$ is such that the following convergence criterion is satisfied, then go to 11.

$$\sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k+1)}))}{L}} - \sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k)}))}{L}} < \textit{Tolerance} .$$

Otherwise, set $k=k+1$, go to 2.

10. Set $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)}$ go to 2.

11. Set $\mathbf{w}_0 = \mathbf{w}^{(k+1)}$ and stop.

3.5 Implementation of the Levenberg-Marquardt Method

The Levenberg-Marquardt technique is used to improve the stability of the Gauss-Newton method, as we have seen in Chapter 2. Specifically, instead of solving a system of linear equation given by

$$(3.5.1) \quad (\mathbf{J}^{(k)T} \mathbf{J}^{(k)}) \mathbf{p}^{(k)} = -\mathbf{g}^{(k)} \quad (\text{step 4, Algorithm 3.4.3}),$$

in the treatment of Levenberg and Marquardt, we solve the system determined by

$$(3.5.2) \quad (\mathbf{J}^{(k)T} \mathbf{J}^{(k)} + \lambda^{(k)} \mathbf{D}) \mathbf{p}^{(k)} = -\mathbf{g}^{(k)},$$

where $\mathbf{D} = \text{diag}(Q_{11}, \dots, Q_{nn})$ for $\mathbf{Q} = \mathbf{J}^{(k)T} \mathbf{J}^{(k)}$. Based on *Algorithm 2.7.1* and *Algorithm 3.4.3*, the Levenberg-Marquardt version of the Gauss-Newton method is straightforward.

Algorithm 3.5.1: Given a set $\mathbf{S} = \{(\mathbf{x}_i, \mathbf{y}_i) \mid \mathbf{x}_i = \text{input}, \mathbf{y}_i = \text{desired output of } \mathbf{x}_i\}$ of L training patterns and given a network setup of K layers with input dimension of n and output dimension of m .

1. Initialize all weights $w_{j,i}^{[k]}$ as random numbers uniformly distributed between

$$\frac{-0.5}{\text{fan-in of that unit}} \quad \text{and} \quad \frac{0.5}{\text{fan-in of that unit}}.$$

set stopping *Tolerance*

set $\mathbf{w}^{(1)} = (w_{j,i}^{[k]})$, $k=1$, $\lambda^{(k)}=0.01$, and $v=10$.

2. For each pattern $(\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{S}$, repeat step 2.1, 2.2, and, 2.3, with $\mathbf{g}^{(k)}=0$.
 - 2.1. Using the weight $\mathbf{w}^{(k)}$, compute the actual output of the network by using (3.1.3) and (3.1.4).
 - 2.2. Obtain the gradient $\mathbf{g}(\mathbf{x}_i)$ of $E(\mathbf{w})$ via the use of (3.4.4), (3.4.5), and (3.3.8),
 - 2.3. sum up $\mathbf{g}(\mathbf{x}_i)$'s, i.e., $\mathbf{g}^{(k)} = \mathbf{g}^{(k)} + \mathbf{g}(\mathbf{x}_i)$
3. For each pattern $(\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{S}$, repeat step 3.1 and 3.2 with $\mathbf{J}^{(k)}=0$.
 - 3.1. Compute the Jacobian Matrix $\mathbf{J}(\mathbf{w}^{(k)}, \mathbf{x}_i)$ via (3.4.8).
 - 3.2. sum up $\mathbf{J}(\mathbf{w}^{(k)}, \mathbf{x}_i)$'s, i.e., $\mathbf{J}^{(k)} = \mathbf{J}^{(k)} + \mathbf{J}(\mathbf{w}^{(k)}, \mathbf{x}_i)$.
4. Set $\mathbf{H}^{(k)} = \mathbf{J}^{(k)T} \mathbf{J}^{(k)}$ and $\lambda^{(k)} = \frac{\lambda^{(k)}}{v}$.
5. Compute $\mathbf{p}^{(k)}$ by solving the system of linear equations
$$(\mathbf{H}^{(k)} + \lambda^{(k)} \mathbf{D}) \mathbf{p}^{(k)} = -\mathbf{g}^{(k)}.$$
6. Set $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{p}^{(k)}$.
7. If $E(\mathbf{w}^{(k+1)}) \geq E(\mathbf{w}^{(k)})$, then set $\lambda^{(k)} = \lambda^{(k)} v$ and go to 5. Otherwise, goto 8.

8. If all the weights $\mathbf{w}^{(k+1)} = (\mathbf{w}_{j,i}^{[k']})^{(k+1)}$ is such that the following convergence criterion is satisfied, then go to 9.

$$\sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k+1)}))}{L}} - \sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k)}))}{L}} < \textit{Tolerance} .$$

Otherwise, set $k=k+1$, goto 2.

9. Set $\mathbf{w}_0 = \mathbf{w}^{(k+1)}$ and stop.

4. THE DAMPED NEWTON LEARNING ALGORITHM

In this Chapter, we develop the proposed ANN learning algorithm. This includes its foundation and the algorithmic implementation. In Section 2.7, we have seen that the Levenberg-Marquardt method is a very effective technique to improve the stability of the Gauss-Newton method. In this Chapter, we incorporate such a technique into Newton's method in order to improve the convergence properties of the Newton method, and implement this new method as an ANN learning law for training feedforward neural networks. The two sections, Section 4.1 and 4.2, can be treated as extensions of Section 2.7 of Chapter 2 and Section 3.5 of Chapter 3 respectively.

4.1 The Damped Newton Method

When Newton method is used for solving non-linear least squares problems, it suffers the same deficiencies as the Gauss-Newton method we have discussed in Section 2.7 and it may also fail because of non-positive definiteness of the Hessian matrix of $E(\mathbf{w})$. To improve the stability of the Newton method, we incorporate the Levenberg-Marquardt technique into Newton's method. Recall that, at each iteration in the Newton method, we determine a downhill step $\Delta^{(k)}$ by solving the system of linear equations given by

$$(4.1.1) \quad \mathbf{H}^{(k)} \Delta^{(k)} = -\mathbf{g}^{(k)},$$

where $\mathbf{g}^{(k)}$ is the gradient of the error function $E(\mathbf{w})$ and $\mathbf{H}^{(k)}$ is the Hessian matrix of $E(\mathbf{w})$. To overcome the singular and non-positive definite problems of the matrix $\mathbf{H}^{(k)}$ that may exist, we solve the following system of linear equations instead.

$$(4.1.2) \quad (\mathbf{H}^{(k)} + \lambda^{(k)} \mathbf{D}) \Delta^{(k)} = -\mathbf{g}^{(k)},$$

where $\lambda^{(k)} > 0$ and \mathbf{D} is a diagonal matrix whose elements are the absolute values of the diagonal elements of $\mathbf{H}^{(k)}$.

The theoretical foundation for the above formulation is related to the three theorems of Section 2.7. In the three theorems, it is assumed that the matrix \mathbf{Q} being symmetric and positive semidefinite. The latter condition is violated in the above formulation since the Hessian matrix could be non-positive definite. By adding a positive number to each of the diagonal elements of the Hessian matrix \mathbf{H} , the resulting matrix could be made positive definite if the added constants are sufficiently large. Hence, upon the restriction that λ is sufficiently large, the three theorems in section 2.7 can still be used to justify the formulation (4.1.2). All results obtained in the discussion of the Gauss-Newton/Levenberg-Marquardt method (Section 2.7) can now be applied to this extended method. This modified version of the Newton method might be called the “Extended” Levenberg-Marquardt method, or the Damped Newton method.

In the next section, we develop an arithmetic implementation of this Damped Newton method as an ANN learning algorithm for training feedforward neural networks.

4.2 The Damped Newton Learning Algorithm

To implement the damped Newton method as an ANN learning law, we need to compute the gradient \mathbf{g} and the Hessian matrix \mathbf{H} of the error function $E(\mathbf{w})$. Based on our previous work on such computations, a slight modification of the *Algorithm 3.5.1* would yield our goal algorithm.

The Damped Newton Learning Algorithm

Algorithm 4.2.1: Given a set $\mathbf{S}=\{(\mathbf{x}_i, \mathbf{y}_i) \mid \mathbf{x}_i = \text{input}, \mathbf{y}_i = \text{desired output of } \mathbf{x}_i\}$ of L training patterns and given a network setup of K layers with input dimension of n and output dimension of m .

1. Initialize all weights $w_{j,i}^{[k]}$ as random numbers uniformly distributed between

$$\frac{-0.5}{\text{fan-in of that unit}} \text{ and } \frac{0.5}{\text{fan-in of that unit}}.$$

set stopping *Tolerance*

set $\mathbf{w}^{(1)} = (w_{j,i}^{[k]})$, $k=1$, $\lambda^{(k)}=0.01$, and $v=10$.

2. For each pattern $(\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{S}$, repeat step 2.1, 2.2, and 2.3, with $\mathbf{g}^{(k)}=0$.
 - 2.1. Using the weight $\mathbf{w}^{(k)}$, compute the actual output of the network by using (3.1.3) and (3.1.4).
 - 2.2. Obtain the gradient $\mathbf{g}(\mathbf{x}_i)$ of $E(\mathbf{w})$ via the use of (3.4.4), (3.4.5), and (3.3.8),
 - 2.3. sum up $\mathbf{g}(\mathbf{x}_i)$'s, i.e., $\mathbf{g}^{(k)} = \mathbf{g}^{(k)} + \mathbf{g}(\mathbf{x}_i)$
3. For each pattern $(\mathbf{x}_i, \mathbf{y}_i) \in \mathbf{S}$, repeat step 3.1 and 3.2 with $\mathbf{H}^{(k)} = 0$.
 - 3.1. Use Algorithm 3.4.2 to compute the Hessian matrix $\mathbf{H}(\mathbf{w}^{(k)}, \mathbf{x}_i)$ of $E(\mathbf{w})$.
 - 3.2. Sum up $\mathbf{H}(\mathbf{w}^{(k)}, \mathbf{x}_i)$'s, i.e., $\mathbf{H}^{(k)} = \mathbf{H}^{(k)} + \mathbf{H}(\mathbf{w}^{(k)}, \mathbf{x}_i)$.
4. Set $\lambda^{(k)} = \frac{\lambda^{(k)}}{v}$.
5. Compute $\mathbf{p}^{(k)}$ by solving the system of linear equations

$$(\mathbf{H}^{(k)} + \lambda^{(k)} \mathbf{D}) \mathbf{p}^{(k)} = -\mathbf{g}^{(k)}.$$
6. Set $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{p}^{(k)}$.
7. If $E(\mathbf{w}^{(k+1)}) \geq E(\mathbf{w}^{(k)})$, then set $\lambda^{(k)} = \lambda^{(k)} v$ and go to 5.

Otherwise, goto 8.
8. If all the weights $\mathbf{w}^{(k+1)} = (w_{j,i}^{[k]})^{(k+1)}$ is such that the following convergence criterion is satisfied, then go to 9.

$$\sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k+1)}))}{L}} - \sqrt{\frac{\sum_{l=1}^L (E(\mathbf{x}_l, \mathbf{w}^{(k)}))}{L}} < \textit{Tolerance} .$$

Otherwise, set $k=k+1$, goto 2.

9. Set $\mathbf{w}_0 = \mathbf{w}^{(k+1)}$ and stop.

To test our proposed learning algorithm, we will implement it using the FORTRAN language. Some other algorithms will also be implemented in order to assess the performance of the new algorithm. The next chapter deals with those implementations and tests.

5. IMPLEMENTATION AND TEST RESULTS

5.1 Language Implementations

To test the new damped Newton learning algorithm, we implement it using the standard FORTRAN 77 language. To compare its performance in enforcing the learning process of ANNs with that of some other methods, the steepest descent learning algorithm and the Gauss-Newton/Levenberg-Marquardt learning algorithm are also programmed. Their names are STEDES for the steepest descent learning algorithm, GNLM for the Gauss-Newton/Levenberg-Marquardt learning algorithm, and NLMD for the Damped Newton learning algorithm.

The network structure of an ANN is represented by two two-dimensional arrays. The first one is array LAYER, which contains information such as the number of layers in the network, the number of nodes in each layer, and some indexing information. One problem encountered in the implementation is the indexing of neurons and weights. In the program, the following indexing formulas are used.

- The index of neuron i in layer k is $\left(\sum_{s=0}^{k-1} \text{number of nodes in layer } S \right) + i$.

Note that the first summation in the above formula depends on the layer only, and for convenience, the summation for each layer is computed early and stored in the second row of array LAYER. Note also that neuron number 0 in each layer is the bias node.

- The index of a connection weight $w_{j,i}^{[k]}$ is defined by

$$\left(\sum_{s=1}^{k-1} \text{number of node in layer } s \cdot (\text{number of node in layer } (s-1)+1) \right) + (j-1) \cdot \text{number of nodes in layer } (k-1) + i$$

The first summation also depends on the layer only and the values are computed and stored in the third row of the array LAYER.

The second array representing a neural network is a two-dimensional array, NEURON, whose first dimension is over the nodes index. It stores the output $u_j^{[k]}$ of each neuron, the first derivative $(u_j^{[k]})'$ of the activation function, and the partial derivatives of the form $\frac{\partial E}{\partial u_j^{[k]}}$. The storage requirement of NEURON is of order $O(n)$, where n is the total number of nodes in the network (including input and bias nodes).

All the weights are stored in the first row of the two-dimensional array WEIGHT with the first dimension over the weight index. The second and the third rows of WEIGHT are used to store the gradient vector of the error function with regard to a pattern and that of the error function over all the patterns respectively. This array takes storage of order, at most, $O(mK)$, where m is the maximum number of nodes in a layer and K is the number of weighted layers in the network. For the steepest descent learning algorithm, the above storage are all that it needs. For the Gauss-Newton learning algorithm, more rows of the array WEIGHT are needed to store the Jacobian matrix of $E(\mathbf{w})$, and a storage for the Hessian matrix of $E(\mathbf{w})$ is also required. The storage requirement of the Hessian matrix is of order $O(m^2K^2)$, with m and K being defined above. For the Damped Newton learning algorithm, the array HESIAN for the Hessian matrix of $E(\mathbf{w})$ is needed and, in addition, a

storage of order $O(n^2)$ is needed to store information of the form $\frac{\partial u_j^{[k_1]}}{\partial u_i^{[k_2]}}$, $\frac{\partial v_j^{[k_1]}}{\partial v_i^{[k_2]}}$, and

$\frac{\partial E}{\partial v_h^{[k_1]} \partial v_j^{[k_2]}}$. The latter array, UUVV, is also needed for the Gauss-Newton/Levenberg-

Marquardt learning algorithm. In the program, other derivatives whose storage are not allocated are computed when needed. The formulas for computing those pieces of information are listed below.

- 1) $(u_j^{[k]})^i = (u_j^{[k]})' w_{j,i}^{[k]}$
- 2) $(u_j^{[k]})^{(i)} = (u_j^{[k]})' u_i^{[k-1]}$
- 3) $(u_j^{[k]})'' = (u_j^{[k]})' (1 - 2u_j^{[k]})$

The program is divided into functional subroutines. The major subroutines are the following.

INITWT () --initialize all connection weights.

NETOUT () --compute the outputs of all neurons in the network upon an input.

COMGRA () --calculate the gradient of $E(\mathbf{w})$ with regard to an learning pattern.

DEU () --compute the derivatives of the form $\frac{\partial E}{\partial u_j^{[k]}}$.

FIDITV () --find an interval for a line search.

QINTER () --perform quadratic interpolation to locate a minimizer (*Algorithm 2.1.2*).

STEDES () --implement the steepest descent learning algorithm (*Algorithm 3.4.1*).

RR () --compute the squared sum function value upon a learning example.

DUVVV () --compute derivatives of the form $\frac{\partial u_j^{[k_1]}}{\partial u_i^{[k_2]}}$ and $\frac{\partial v_j^{[k_1]}}{\partial v_i^{[k_2]}}$.

COMJAC () --compute the Jacobian matrix of the network function and approximate the Hessian matrix by the product of the Jacobian matrices.

LSOLV () --solve a system of linear equations (*Algorithm 2.1.1*).

DEVV () --compute derivatives of the form $\frac{\partial E}{\partial v_{j_1}^{[k_1]} \partial v_{j_2}^{[k_2]}}$.

COMHAS () --compute the Hessian matrix of $E(\mathbf{w})$ for the Damped Newton learning algorithm.

GNLMD () --implement the Gauss-Newton/Levenberg-Marquardt learning algorithm (*Algorithm 3.5.1*).

NLMD () --implement the Damped Newton learning algorithm (*Algorithm 4.2.1*).

The program is written in FORTRAN 77 with double precision. It is compiled by using ptx/FORTRAN compiler which is compatible with ANSI Standard FORTRAN 77. The test results are described in section 5.3.

5.2 Neural Network Design

The design of a neural network is highly problem-oriented. It is the problem that determines what ANN topology and what stopping condition for training should be used. The topology of the ANN determines the number of connection weights. Using more connection weights implies that the ANN might need more training examples. If, in the training process, the error function values converge to a value above the requirement,

then more connection weights (by means of adding more nodes and/or hidden layers) are needed in order to further reduce the error function values.

Normally, for a given problem, training is done on different topologies of ANNs in order to determine which one might fit the requirements of the problem. At this stage, the stopping conditions used in the training might be any of the stopping condition 2, 3, 4 or 6 as mentioned in section 3.2, Chapter 3. Once a set of optimal ANN topologies are obtained, then the training of the ANNs are concentrated on the generalization of the ANNs. The generalization ability of the ANN is highly sensitive to when to stop the training process. A good fitting of the training examples does not mean that the ANN will generalize well over the entire problem domain. Thus, to obtain better generalization results, some optimal stopping point has to be set before the training starts. The uses of validation sets provide some tools for obtaining such a optimal point. In such a training paradigm, the training process is stopped when the errors on the validation examples are reduced within tolerance. In this stage, the stopping condition used for the training might be the stopping condition 1, 3, or 5. However, in this paper, we are not concerned with the generalization of the ANNs, since it depends on the training ability of the learning algorithms, i.e., the ability to reduce the error function values to within a specified level of tolerance. Hence, to test our new Damped Newton learning algorithm, we emphasize on the training side, that is, whether it trains an ANN or not.

5.3 Test Problems

The goal of our testing is to explore the feasibility of the proposed new damped Newton learning algorithm for training fully-connected, feedforward neural networks and, if feasible, to assess its performance compared with existing learning algorithms.

The first test problem is the parity problem obtained from the benchmark problems for training neural networks in the artificial intelligence depository at Carnegie Mellon University (Anonymous FTP: /afs/cs/project/connect/bench on ftp.cs.cmu.edu). We test the learning algorithms on three sub-problems of the parity problem, i.e., the 2-, 3-, and 4-parity problems. Details of those problems follow.

Problem	Type of Problems	Inputs	Outputs	Number of Examples
2-parity(XOR)	Classification	2	1	4
3-parity	Classification	3	1	8
4-parity	Classification	4	1	16

Note that in this n -parity problem, the requirement is that the ANN should be able to classify each pattern correctly up to a given tolerance. Hence, all the examples are used in the training sessions and generalization of the ANN is not a concern in this problem.

Three more test problems are chosen from the PROBEN1 [26]. Two of those are function approximation problems and the other is a classification problem. The following lists information about those test problems.

Problem	Type of Problems	Inputs	Outputs	Number of Examples
cancer	Classification	9	2	699
building	Approximation	14	3	4208
heart	Approximation	35	1	920

5.4 Test Results

The initial set of connection weights in an ANN affects the learning process of the neural network. Hence, it is necessary to obtain testing results with regard to different set of initial connection weights. The test results obtained in this section are the results of several runs, each time with a new set of random connection weights. In the program, a pseudo-random number generator [32] is used to produce the random numbers needed. The generator takes an integer seed number as its input and could produce a sequence of random numbers unique to each seed number. This way, the weight initialization process could be reproduced so that we can start different learning algorithms with the same ANN topology and the same set of initial connection weights.

The program has been run on the test problems and numerical results have been obtained. The testing results are described below according to the following comparison criteria.

1. Feasibility test: with a fixed stopping criterion, the program was run on the three test problems. For each problem, ten runs have been performed for each of the three learning algorithms. The initial weights of the network are different for each run, while they are the same for each of the three learning algorithms at each run (i.e., using the same seed for generating the pseudo-random numbers). The results obtained are the average of the ten runs or fewer if learning fails because of local minimum problems.

2. Performance test: the same as above but with varied stopping conditions to assess the performance of the new algorithm when high precision over the training set is needed.

The following lists the parameters used in the three learning algorithms.

- initial $\lambda=0.01$
- increase/decrease factor $\mu=5.0$

The feasibility test has been done on all of the test problems mentioned in the last section. For the classification problems, the stopping condition used is No. 5, i.e., to classify each pattern correctly to within a 0.1 tolerance. The results are listed in the following tables. In the tables, the symbol “L” means a local minimum was encountered, i.e., patterns can not all be classified correctly within the given tolerance, and “F” implies that the method failed. In such cases, they are not counted in the averages.

Table 1: Test Results on the 2-parity Problem with a 2-4-1 ANN Topology

	seed number for generating initial random weights										average
Algorithm	1	17	21	27	40	45	66	78	81	96	
Steepest Descent	80	61	68	49	106	103	104	106	35	259	97.1
Gauss-Newton/LM	6	9	6	L	6	15	5	L	2	11	7.5
Damped Newton	21	L	9	15	18	16	17	37	17	19	18.8

Note: 4 training examples are used.

Table 2: Test Results on the 3-parity Problem with a 3-6-1 ANN Topology

	seed number for generating initial random weights										average
Algorithm	1	17	21	27	40	45	66	78	81	96	
Steepest Descent	135	402	3660	5541	11821	480	3401	149	4696	189	3047.4
Gauss-Newton/LM	16	6	3	7	5	6	10	12	7	9	8.1
Damped Newton	L	28	12	14	28	16	L	15	19	23	19.4

Note: 8 training examples are used.

Table 3: Test Results on the 3-parity Problem with a 4-8-1 ANN Topology

	seed number for generating initial random weights										average
Algorithm	1	17	21	27	40	45	66	78	81	96	
Steepest Descent	1121	1334	18576	4325	1489	2783	2811	F	23230	2242	3617.2
Gauss-Newton/LM	L	L	105	132	L	L	119	101	18	57	88.7
Damped Newton	21	116	22	34	54	L	77	59	55	119	61.4

Note: 16 training examples are used.

The stopping conditions used for testing the cancer classification problem and the other two approximation problems are somewhat complicated. We first run the Gauss-Newton/Levenberg-Marquardt learning algorithm (or the Damped Newton) with a relative RMS stopping condition (Stopping Condition #3) to determine a point of convergence. Note that this stopping condition sometimes does not work well when used in the steepest descent learning algorithm, since the reduction of function values from one iteration to the next could be very small. Then, upon the knowledge of this point, we used the stopping condition #1, i.e., learning is stopped when the error function values are reduced within a given tolerance (which is greater than the function value at the point of

convergence). In the following, results of the Steepest Descent learning algorithm are obtained with a tolerance of 0.001 with stopping condition #1 (using RMS values) and the results of the other two methods are gotten when the reduction of RMS values is within tolerance of 0.1×10^{-16} (Stop Condition #3).

Table 4: Test Results on the Cancer Problem with a 9-2 ANN Topology

Algorithm	seed number for generating initial random weights										average
	1	17	21	27	40	45	66	78	81	96	
Steepest Descent	2173	2184	2190	2190	1924	2037	2165	2130	346	2180	1951.9
Gauss-Newton/LM	58	57	58	56	56	56	56	57	56	56	56.6
Damped Newton	12	12	12	12	12	12	13	12	12	12	12.1

Note: (1) 525 training examples are used.

(2) Local minimum=0.2246622184498266780 (with a 10^{-15} accuracy).

(3) Training for Steepest Descent algorithm stopped when function values are reduced below 0.231.

Table 5: Test Results on the Building Problem with a 14-3 ANN Topology

Algorithm	seed number for generating initial random weights										average
	1	17	21	27	40	45	66	78	81	96	
Steepest Descent	748	757	757	771	753	760	773	763	747	767	759.6
Gauss-Newton/LM	7	7	7	7	7	7	7	7	7	7	7
Damped Newton	6	6	6	6	6	6	5	5	6	6	5.8

Note: (1) 500 training examples are used.

(2) Local minimum=0.0602037866342696670 (with a 10^{-15} accuracy).

(3) Training for Steepest Descent algorithm stopped when function values are reduced below 0.0603.

Table 6: Test Results on the Heart Problem with a 35-1 ANN Topology

Algorithm	seed number for generating initial random weights										average
	1	17	21	27	40	45	66	78	81	96	
Steepest Descent	325	341	331	336	324	325	322	327	335	335	330.1
Gauss-Newton/LM	9	10	9	9	9	9	9	9	9	9	9.1
Damped Newton	5	5	5	5	5	5	6	5	5	6	5.2

Note: (1) 690 training examples are used.

(2) Local minimum=0.1970472022241477990 (with a 10^{-15} accuracy).

(3) Training for Steepest Descent algorithm stopped when function values are reduced below 0.198.

One advantage of using the Newton method over the Gauss-Newton method is that the former converges to a minimum point with quadratic convergence as opposed to the linear convergence for the Gauss-Newton method. When testing, this property is reflected by the speed at which the norm of the weight updates approaches 0. For the damped Newton method, its speed of convergence is weaker than that of the Newton method and is close to quadratic as can be observed in the following tables. The stopping condition used for the tests is the relative RMS (Stopping Condition #3) with a much smaller tolerance of 0.1×10^{-16} than that used in the above tests. This stronger requirement is needed when high precision on the training set is required.

Table 7: Performance Test Results (the Gauss-Newton/Levenberg-Marquardt Algorithm) on the Building Problem with a 14-3 ANN Topology

actual RMS	epoch	$\lambda^{[k]}$	$\ \Delta w\ _2$
0.3719656416874160150	0	0.1000000E-01	5.000000
0.0764699001570153846	1	0.2000000E-02	2.851237
0.0603818254426006184	2	0.4000000E-03	0.6581495
0.0602039906224533627	3	0.8000000E-04	0.1705101
0.0602037872064130574	4	0.1600000E-04	0.8345467E-02
0.0602037866363822171	5	0.3200000E-05	0.4318909E-03
0.0602037866342778116	6	0.6400000E-06	0.2682336E-04
0.0602037866342696670	7	0.1280000E-06	0.1654361E-05
0.0602037866342696492	8	0.2560000E-07	0.1031173E-06

Note: (1) 500 training examples are used.

Table 8: Performance Test Results (the Damped Newton Algorithm) on the Building Problem with a 14-3 ANN Topology

actual RMS	epoch	$\lambda^{[k]}$	$\ \Delta w\ _2$
0.3719656416874160150	0	0.1000000E-01	5.000000
0.0787833370965010093	1	0.2000000E-02	2.805360
0.0613366701825997751	2	0.4000000E-03	0.9584970
0.0602151588752766375	3	0.8000000E-04	0.4732735
0.0602037887783644087	4	0.1600000E-04	0.8338464E-01
0.0602037866342702443	5	0.3200000E-05	0.1488150E-02
0.0602037866342696847	6	0.6400000E-06	0.1317166E-05
0.0602037866342696670	7	0.1280000E-06	0.2642247E-09

Note: (1) 690 training examples are used.

Table 9: Performance Test Results (the Gauss-Newton/Levenberg-Marquardt Algorithm) on the Heart Problem with a 35-1 ANN Topology

actual RMS	epoch	$\lambda^{(k)}$	$\ \Delta w\ _2$
0.3091762029727637360	0	0.1000000E-01	5.000000
0.2000977683803623730	1	0.2000000E-02	3.142956
0.1970759648661704050	2	0.4000000E-03	0.7461564
0.1970474538977780020	3	0.8000000E-04	0.9303208E-01
0.1970472055096417030	4	0.1600000E-04	0.8621738E-02
0.1970472022720594110	5	0.3200000E-05	0.8791360E-03
0.1970472022248880070	6	0.6400000E-06	0.9898951E-04
0.1970472022241596290	7	0.1280000E-06	0.1136560E-04
0.1970472022241478880	8	0.2560000E-07	0.1339874E-05
0.1970472022241477100	9	0.5120000E-08	0.1799136E-06
0.1970472022241478520	10	0.5120000E-09	0.1718461E-05
0.1970472022241476390	10	0.9765625E-01	0.1655321E-07

Note: (1) 500 training examples are used.

Table 10: Performance Test Results (the Damped Newton Algorithm) on the Heart Problem with a 35-1 ANN Topology

actual RMS	epoch	$\lambda^{(k)}$	$\ \Delta w\ _2$
0.3091762029727637360	0	0.1000000E-01	5.000000
0.2000021570874490610	1	0.2000000E-02	3.194229
0.1971521541708801450	2	0.4000000E-03	0.7650341
0.1970474745554628800	3	0.8000000E-04	0.1775413
0.1970472022267991890	4	0.1600000E-04	0.9683954E-02
0.1970472022241477280	5	0.3200000E-05	0.3717560E-04
0.1970472022241477460	6	0.6400000E-06	0.6030849E-08
0.1970472022241477100	6	0.1280000E-06	0.6028837E-08

Note: (1) 690 training examples are used.

6. CONCLUSION

6.1 Conclusion

This study presents a new ANN learning algorithm for training fully-connected, feedforward neural networks. The new algorithm is based on Newton's method for solving non-linear least squares problems and the Levenberg-Marquardt technique to improve the convergence properties of Newton's method. The new algorithm has been programmed and tested on several real world problems. Satisfactory numerical results have been obtained. It is shown in this paper that the proposed new learning algorithm is practically feasible and has high convergence performance over the existing ANN learning algorithms when dealing with approximation problems and when high precision over the training set is required.

6.2 Recommendation for Future Work

Further investigation could be done at several places in the new algorithm. Some optimal methods could be used in order to improve the performance of the new algorithm. Those areas are described in the following.

1. Further investigation can be done on some optimal choices of λ and the factor μ at each iteration, which are used in both of the Gauss-Newton/Levenberg-Marquardt learning algorithm and the Damped Newton learning algorithm. One approach is that of Fletcher [5]. Others are covered in references [12], [13], [17] and [33].

2. Other paradigms regarding when and how to increase or decrease λ at each iteration other than that of Marquardt's approach presented in this paper can be investigated. The references are the same as the above ones.
3. A rigid comparison study is needed to compare thoroughly the performance of the algorithms mentioned in this paper in order to obtain better understanding of the learning properties of those algorithms we are concerned with, with regard to more aspects of the learning process in training ANNs. For example, the Gauss-Newton method may never have a neighborhood of convergence at a minimum point, as has been shown in [31]. Such a problem is not encountered in the test problems we have seen in this thesis report. Proper test problems need to be found in order to gain understanding of this divergence behavior of the Gauss-Newton method and the behavior of Newton's method when tested with this kind of problems.

7. REFERENCES

- [1] M. Al-Baali and R. Fletcher, "Variational Methods for Non-Linear Least-Squares", *J. Opl Res. Soc.*, Vol. 36, No.5, pp405-421, 1985.
- [2] Christine T. Altendorf, *Estimating Soil Water Content Using Soil Temperatures and a Neural Network*, Ph.D. Dissertation, Oklahoma State University, 1993
- [3] Roberto Battiti, "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method", *Neural Computation*, Vol.4, 1992, pp141-166.
- [4] Randall Beer, Chiel, Hillel; and Sterling, Leon, "An Artificial Insect", *American Scientist*, Volume 79.
- [5] Chris Bishop, "A Fast Procedure for Retraining the Multilayer Perceptron", *International Journal of Neural Systems*, Vol. 2, No. 3 (1991), pp229-236.
- [6] Chris Bishop, "Exact Calculation of the Hessian Matrix for the Multilayer Perceptron", *Neural Computation*, Vol.4, 1992, pp494-501.
- [7] Chris M. Bishop, "Neural Networks and Their Applications", *Rev. Sci. Instrum.* Vol.65, No. 6, June, 1994.
- [8] Wray L. Buntine and Andreas S. Weigend, "Computing Second Derivatives in Feed-Forward Networks: A Review", *IEEE Transactions on Neural Networks*, Vol. 5, No.3, pp.480-488, May, 1994.
- [9] Richard L. Burden and J. Douglas Faires, *Numerical Analysis*, 4th Edition, PWS-KENT Publishing Company, Boston, 1988.

- [10] A. Cichocki and R. Unbehauen, *Neural Networks for Optimization and Signal Processing*, John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1993.
- [11] J. E. Dennis, Jr. and Robert B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1983.
- [12] A. Dold and B. Eckmann, *Lecture Notes in Mathematics: Numerical Analysis*, No. 630, Springer-Verlag, Berlin, 1978.
- [13] R. Fletcher, *Practical Methods of Optimization*, Volume 1, John Wiley & Sons, Chichester, 1980.
- [14] A. Forsgren, P.E. Gill, and W. Murray, "Computing Modified Newton Directions Using A Partial Cholesky Factorization", *SIAM J. SCI. COMPUT.*, Vol.16, No.1, pp139-150, January 1995.
- [15] Martin T. Hagan and Mohammed B. Menhaj, "Training Feedforward Networks with the Marquardt Algorithm", *IEEE Transactions on Neural Networks*, Vol. 5, No.6, pp.989-994, Nov.,1994.
- [16] Martin T. Hagan, *Neural Networks*, Lecture Notes, Oklahoma State University, 1995.
- [17] W. M. Haubler, "A Local Convergence Analysis for the Gauss-Newton and Levenberg-Morrison-Marquardt Algorithm", *Computing*, Vol.31, pp231-244, 1983.

- [18] Magnus R. Hestenes, *Conjugate Direction Methods in Optimization*, Springer-Verlag, 175 Fifth Avenue, New York, NY 10010, USA, 1980.
- [19] Robert Hecht-Nielsen, *Neurocomputing*, Addison-Wesley Publishing Company, Inc., 1990.
- [20] Nicolaos B. Karayiannis and Anastasios N. Venetsanopoulos, *Artificial Neural Networks*, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, 02061, 1993.
- [21] S.Y. Kung, *Digital Neural Network*, Princeton University, PTR Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1993.
- [22] Charles L. Lawson and Richard J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.
- [23] Kenneth Levenberg, "A Method For the Solution of Certain Non-Linear Problems in Least Squares", *Quart. Appl. Math.*, No.2, pp.164-168, 1944.
- [24] Donald W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters", *J. Soc. Indust. Appl. Math.*, Vo.11, No.2, pp.431-441, June, 1963
- [25] Martin Fodslette Moller, "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning", *Neural Networks*, Vol. 6, pp. 525-553, 1993.
- [26] Lutz Prechelt, *PROBEN1-A Set of Neural Network Benchmark Problems and Benchmarking Rules*, Technical Report 21/94, University Karlsruhe, 1994, Anonymous FTP: /pub/neuron/proben1.tar.gz on ftp.ira.uka.de.
- [27] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, "Learning representations by back-propagating errors", *Nature*, Vol.323-9, Oct., 1986.

- [28] L.E. Scales, *Introduction to Non-Linear Optimization*, Springer-Verlag New York Inc., 175 Fifth Avenue, New Your, NY 10010, USA, 1985.
- [29] F. Beckley Smith, Jr., and David F. Shanno, "An Improved Marquardt Procedure for Nonlinear Regressions", *Technometrics*, Vol.13, No.1, pp.63-74, Feb., 1971.
- [30] J.G. Taylor, *The Promise of Neural Networks*, Springer-Verlag, London, 1993.
- [31] D. R. Unruh, *Basic Fluid Power Research Program*, OSU Report, REF 70-1, Oklahoma State University, 1970.
- [32] Shiang-Huey Wang, "Comparison of Backpropagation Neural Networks and General Regression Neural Networks", *M.S. Thesis*, Oklahoma State University, 1994.
- [33] Andrew R. Webb, "Functional Approximation by Feed-Forward Networks: A Least-Squares Approach to Generalization", *IEEE Transactions on Neural Networks*, Vol. 5, No. 3, pp363-371, May, 1994.
- [34] Yuan Wei, "An Accelerated Levenberg-Marquardt Algorithm for Nonlinear Least Square Problems", *M.S. Thesis*, Oklahoma State University, 1992.
- [35] Patrick H. Winston, *Artificial Intelligence*, Third Edition, Addison-Wesley Publishing company, Reading, Massachusetts, 1992.
- [36] M.A. Wolfe, *Numerical Methods for Unconstrained Optimization*, Van Nostrand Reinhold Company Ltd., Molly Millars Lane, Wokingham, Berkshire, England, 1978.
- [37] FAQ, comp.ai.neural-nets, URL: <http://www.ipd.ira.uka.de/~prechelt/FAQ/neural-net-faq.html>.

8. APPENDICES

8.1 Program List

```
PROGRAM DRIVER
C*****
C
C BY Joseph Wang, Department of Computer Sciences, OSU, 1995
C
C*****
C This is the driver of the 3 ANN learning algorithms implemented.
C Before the program is executed, the following 3 data files should be
C set ready. Note that if there are more than one data in a line than
C they should be separated by spaces or commas.
C   (1). "net.dat", contains information to specify a network.
C       FORMAT: 1st line--number of weighted layers (without input
C               layer)
C               2nd line--number of nodes in each layer, starting
C               with the input-node layer, all in one
C               line.
C               3rd line--an integer seed for generating random
C               number.
C               4th line--stopping tolerance.
C               5th line--choice of learning algorithms.
C                   1= Steepest Descent.
C                   2= Gauss-Newton/Levenberg-Marquadt
C                     algorithm with Identity matrix added.
C                   3= Damped Newton algorithm with Identity
C                     matrix added.
C                   4= Gauss-Newton/Levenberg-Marquadt
C                     algorithm with diagonal matrix added.
C                   5= Damped Newton algorithm with diagonal
C                     matrix added.
C   (2). "train.dat", contains training examples.
C       FORMAT: 1st line--number of training examples.
C               2nd line--input example, all in one line.
C               3rd line--desired output examples.
C               repeat the pattern of 2nd and 3rd lines for each
C               example.
C   (3). "test.dat", contains test examples
C       FORMAT: 1st line--number of testing examples.
C               2nd line--input values, all in one line.
C               repeat the 2nd lines for each testing example.
C
C When running, one of the following routine will be called.
C   STEDES -- the Steepest Descent learning algorithm
C   GNLM   -- the Gauss-Newton/Levenberg-Marquardt learning algorithm
C   GNLM   -- the Newton/Levenberg-Marquardt learning algorithm
C
C*****
C Important: This program is neural network dependent. That is, the
C program has restriction on the number of neurons in a layer
C and the number of layers in a network. The restriction is
C due to the language deficiency rather than the performance
```

C of the learning algorithms.
 C If it is needed to change the capacity of the program, then
 C the dimensions of all arrays should be adjusted properly.
 C This only needs to be done for the global variables that
 C defined below. The dimensions of the arrays in subprograms
 C will be automatically adjusted by using formal arguments.
 C

C Variables:

C LAYER: to store network structure.
 C 2 dimension integer array, 1st index over layers (including
 C the input nodes layer) of the network, the 2nd index are
 C defined as following:
 C (*, 1) contains the number of nodes in each layer,
 C excluding the bias node.
 C (*, 2) keeps information for indexing neurons in the net.
 C (*, 3) stores information for indexing the weights in the
 C net.

C INEX: to store learning examples.
 C 2 dimension real array, 1st index over input vector, 2nd
 C over learning examples.

C OUTEX: to store the desired outputs of learning examples.
 C 2 dimension real array, 1st index over output vector, 2nd
 C over learning examples.

C TESTEX: to store test examples.
 C 2 dimension real array, 1st index over input vector, 2nd
 C over testing examples.

C NEURON: to store information related to neurons.
 C 2 dimension real array, 1st index over all neurons/nodes in
 C the network, the 2nd index are defined as below
 C (*, 1) contains output values of each neuron/node in the
 C net.
 C (*, 2) contains 1st derivatives of the activation function
 C of each neurons.
 C (*, 3) stores the partials of $E(w)$ w.r.t. a neuron output
 C (*, 4) temporary

C WEIGHT: to store information w.r.t. connection weights.
 C 2 dimension real array, 1st index over all connection
 C weights in the network, the 2nd index are defined as below.
 C Note that the 2nd index should be greater than
 C (net output dim. + 4).
 C (*, 1) contains weight values of all connections.
 C (*, 2) stores the partials of $E(w)$ w.r.t. a weight, i.e.,
 C the gradient of $E(w)$.
 C (*, 3) temporary used by STDDDES, COMJAC.
 C (*, 4) temporary.
 C (*, 5+*) storage for the Jacobian matrix.

C UUVV: to store information w.r.t. 2 neurons.
 C 3 dimension real array, contains useful derivatives
 C (*,*,1) contains partials of the activation function of a
 C neuron w.r.t. the output of another neuron in its
 C input path.
 C (*,*,2) contains partials of the weighted sum function of
 C a neuron w.r.t. the weighted sum of another
 C neuron in its input path.
 C (*,*,3) stores the 2nd order partials of $E(w)$ w.r.t. two
 C weighted sum functions $v(k1,j1)$ and $v(k1,j2)$ with
 C $k1 \geq k2$.
 C (*,*,4) temporary

C HESIAN: the Hessian matrix of $E(w)$.
 C 3 dimensional real array. 1st index (row) and 2nd index
 C forms the indices of the Hessian matrix. 3 storage space

```

C          are used.
C
C  INDIM:  input dimension.
C  OUTDIM: output dimension.
C  NUMEX:  number of learning examples.
C  NUMTEX: number of testing examples.
C  NUMUNI: number of units in the net, including input and bias nodes.
C  WTDIM:  number of weights in the net, dimension of the weight
C          vector.
C  SEED:   SEED of pseudo-random number generator.
C
C
C *****
C  Declaration
C
C  INTEGER LAYER(-1:11,3)
C  DOUBLE PRECISION INEX(35, 800), OUTEX(4,800), TESTEX(35,800),
+          WEIGHT(0:199,10), NEURON(0:49,4),
+          UUVV(0:49,0:49, 4), HESIAN(0:199,0:199,3)
C  INTEGER INDIM, OUTDIM, NUMEX, NUMTEX, WTDIM, SEED, METHOD
C
C  DOUBLE PRECISION TOLERA, STPSIZ
C
C --- The following variables are used to pass the dimension indices
C      when calling subroutines, their values should be the same as the
C      dimensions numbers specified in the above declarations.
C
C  INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, IEIDX1, IEIDX2,
+          OEIDX1, OEIDX2, TEIDX1, TEIDX2, NUIDX1, NUIDX2,
+          UUIDX1, UUIDX2, UUIDX3
C          LAIDX1=11
C          LAIDX2=3
C          IEIDX1=35
C          IEIDX2=800
C          OEIDX1=4
C          OEIDX2=800
C          TEIDX1=35
C          TEIDX2=800
C          WTIDX1=199
C          WTIDX2=10
C          NUIDX1=49
C          NUIDX2=4
C          UUIDX1=49
C          UUIDX2=49
C          UUIDX3=4
C
C --- setup internal network representation
C  CALL SETNET(LAYER, LAIDX1, LAIDX2, SEED, TOLERA, METHOD)
C  INDIM=LAYER(-1,2)
C  OUTDIM=LAYER(-1,3)
C  K=LAYER(-1,1)
C  WTDIM=LAYER(K,3)+LAYER(K,1)*(LAYER(K-1,1)+1)
C  setup parameters
C  CALL SETPAR()
C
C --- read in training examples
C  CALL SETEX(INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
+          INDIM, OUTDIM, NUMEX)
C
C --- check net structure and examples
C  CALL PRTNET(LAYER, LAIDX1, LAIDX2)

```



```

C      CALL PRTEX(INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
C      +          INDIM, OUTDIM, NUMEX)
C
C --- initialize all connection weights
C      read in an integer SEED for generating pseudo-random number
C      PRINT *, 'Input an integer for the random SEED'
C      READ *, SEED
C      CALL INITWT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, SEED)
C
C --- check connection weights
C      CALL PRTWT(LAYER, LAIDX1, LAIDX2, WEIGHT,WTIDX1)
C
C --- read in stopping criterion
C      PRINT *, 'Input RMS tolerance below '
C      READ *, TOLERA
C
C --- choose a learning algorithm
C      GOTO 18
16     WRITE (*, 17)
17     FORMAT(/, 'Choose a training method:',/,6X,
+           '1=steepest descent method',/,6X,
+           '2=Gauss-Newton method-(I)',/,6X,
+           '3=Damped Newton method-(I)',/,6X,
+           '4=Gauss-Newton method-(D)',/,6X,
+           '5=Damped Newton method-(D)')
C      READ (*,*) METHOD
C
18     IF (METHOD .EQ. 1) THEN
C         The following parameter is not adjustable.
C         STPSIZ=1.0
C         CALL STEDES(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                 NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+                 OUTEX, OEIDX1, OEIDX2, NUMEX, WTDIM,
+                 INDIM, OUTDIM, STPSIZ, TOLERA)
C         ELSE IF (METHOD .EQ. 2) THEN
C             CALL GNLM(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                     NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+                     OUTEX, OEIDX1, OEIDX2, NUMEX,
+                     INDIM, OUTDIM, TOLERA,
+                     UVVV, UUIDX1, UUIDX2, UUIDX3,
+                     HESIAN, WTDIM)
C         ELSE IF (METHOD .EQ. 3) THEN
C             CALL NLM(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                    NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+                    OUTEX, OEIDX1, OEIDX2, NUMEX,
+                    INDIM, OUTDIM, TOLERA,
+                    UVVV, UUIDX1, UUIDX2, UUIDX3,
+                    HESIAN, WTDIM)
C         ELSE IF (METHOD .EQ. 4) THEN
C             CALL GNLM(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                     NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+                     OUTEX, OEIDX1, OEIDX2, NUMEX,
+                     INDIM, OUTDIM, TOLERA,
+                     UVVV, UUIDX1, UUIDX2, UUIDX3,
+                     HESIAN, WTDIM)
C         ELSE IF (METHOD .EQ. 5) THEN
C             CALL NLMD(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                     NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+                     OUTEX, OEIDX1, OEIDX2, NUMEX,
+                     INDIM, OUTDIM, TOLERA,
+                     UVVV, UUIDX1, UUIDX2, UUIDX3,

```

```

+          HESIAN, WTDIM)
ELSE
  PRINT *, 'Illegal choice, try again.'
  GOTO 16
ENDIF
C
C --- start testing
C read in testing examples
CALL SETTEX(TESTEX, TEIDX1, TEIDX2, INDIM, NUMTEX)
C
C test network on testing examples
DO 30 J=1, NUMTEX
  DO 20 I=1, INDIM
    NEURON(I,1)=TESTEX(I,J)
20  CONTINUE
    NEURON(0,1)=-1
C compute output for a testing input
CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+          NEURON, NUIDX1, NUIDX2)
C
  WRITE (*,25) (NEURON(LAYER(LAYER(-1,1),2)+I, 1), OUTEX(I, J),
+          I=1, OUTDIM)
25  FORMAT (2(' a=', G15.10, ' d=', G15.10))
30  CONTINUE
C
C --- print out the final set of weights
CALL PRTWT(LAYER, LAIDX1, LAIDX2, WEIGHT,WTIDX1)
C
C*****
  STOP
  END
C*****

C*****
SUBROUTINE SETNET(LAYER, LAIDX1, LAIDX2, SEED, TOLERA, METHOD)
C*****
C This subroutine is to read the input file, containing specification
C of a neural network structure, and setup program parameters
C representing the network structure.
C
  INTEGER LAIDX1, LAIDX2, LAYER(-1:LAIDX1, LAIDX2),SEED,METHOD
  DOUBLE PRECISION TOLERA
  INTEGER IOERR, TEMP1, TEMP2
C
  INUNIT = 50
  OPEN(UNIT= INUNIT, FILE='net.dat', STATUS='OLD', IOSTAT = IOERR)
  IF (IOERR .NE. 0) THEN
    PRINT 110, IOERR
110  FORMAT('can not open net data file (net.dat), IOERR =',I10)
    GOTO 150
  ENDIF
C
C read in the # of layers (excluding input layer of nodes)
READ(INUNIT,*) LAYER(-1,1)
C read in the #'s of nodes in each layer (1st is # of input nodes)
READ(INUNIT,*) (LAYER(I,1), I= 0,LAYER(-1,1))
C
  READ(INUNIT,*) SEED
  READ(INUNIT,*) TOLERA
  READ(INUNIT,*) METHOD

```

```

C      setup input and output dimensions
      LAYER(-1,2)= LAYER(0,1)
      LAYER(-1,3)= LAYER(LAYER(-1,1),1)
C
C      setup parameters for indexing NEURON array and WEIGHT array
      LAYER(0,2)=0
      LAYER(0,3)=0
      LAYER(1,2)=LAYER(0,1)+1
      LAYER(1,3)=0
      TEMP1=LAYER(0,1)+1
      TEMP2=0
      DO 115 I=2, LAYER(-1,1)
          TEMP1=TEMP1+LAYER(I-1,1)+1
          TEMP2=TEMP2+LAYER(I-1,1)*(LAYER(I-2,1)+1)
          LAYER(I,2)=TEMP1
          LAYER(I,3)=TEMP2
115  CONTINUE
C
      CLOSE (UNIT = INUNIT)
C*****
150  RETURN
      END
C*****
C
C*****
      SUBROUTINE PRINET(LAYER, LAIDX1, LAIDX2)
C*****
C This subroutine is to print out the network structure.
C
      INTEGER LAIDX1, LAIDX2, LAYER(-1:LAIDX1, LAIDX2)
C
C      print them out
      PRINT 121,LAYER(-1,1)
      WRITE (*,122) (LAYER(I,1), I=0,LAYER(-1,1))
      WRITE (*,125) (LAYER(I,2), I=0,LAYER(-1,1))
      WRITE (*,126) (LAYER(I,3), I=0,LAYER(-1,1))
      PRINT 123, LAYER(-1,2)
      PRINT 124, LAYER(-1,3)
      K=LAYER(-1,1)
      PRINT 127, LAYER(K,3)+LAYER(K,1)*(LAYER(K-1,1)+1)
121  FORMAT (/, 'The net has ',I2, 1X, 'weighted layers.')
122  FORMAT ('The number of nodes in each layer is',
+          I4, 100(' ', ' ', I4))
123  FORMAT ('The input dimension is ',I4)
124  FORMAT ('The output dimension is ',I4)
125  FORMAT ('The 1st parameter vector=',
+          I4, 100(' ', ' ', I4))
126  FORMAT ('The 2nd parameter vector=',
+          I4, 100(' ', ' ', I4))
127  FORMAT ('The number of connection weights=',
+          I4, 100(' ', ' ', I4))
C
C*****
      RETURN
      END
C*****
C*****
      SUBROUTINE SETPAR()
C*****
C This routine set up all parameters needed in the program

```

```

C
INTEGER LIM1, LIM2, LIM3, LIM4
DOUBLE PRECISION XMAX, XMIN, INILAM, INIFAC, CUTOFF, SMALL
C
COMMON /LIMIT/XMAX, XMIN
COMMON /LIMIT1/LIM1,LIM2,LIM3
COMMON /LIMIT2/LIM4
COMMON /LEVMAR/INILAM, INIFAC, CUTOFF, SMALL
C
XMAX and XMIN is used by NETOUT to compute the output of a neuron.
C
Since the Sigmoid function subroutine will overflow for X too
C
large or too small, XMAX and XMIN are the limits used to set the
C
function values to be 1.0 and 0.0, respectively.
XMAX= 700.0
XMIN=-700.0
C
C
LIM1, LIM2, and LIM3 are used in FIDITV to control the loops
LIM1=50
LIM2=1000
LIM3=100
C
C
LIM4 is used in QINTER to control a loop
LIM4=100
C
C
INILAM and INIFAC are the two initial parameters used for the
C
Levenberg-Marquardt modification of the Gauss-Newton method and
C
the Newton method. INILAM is the initial LAMBDA value and INIFAC
C
is the initial FACTOR value. CUTOFF is used as limit point. When
C
LAMBDA is smaller than the CUTOFF, then LAMBDA is set to SMALL.
INILAM=0.01
INIFAC=5.0
CUTOFF=1.0D-8
SMALL=1.0D-16
C
C*****
RETURN
END
C*****

C*****
SUBROUTINE SETEX(INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
+
INDIM, OUTDIM, NUMEX)
C*****
C This subroutine is to read the input file, containing the training
C
examples(inputs along with desired outputs) and to fill out the
C
example arrays.
C
INTEGER IEIDX1,IEIDX2,OEIDX1,OEIDX2,INDIM, OUTDIM, NUMEX
DOUBLE PRECISION INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2)
C
C
INTEGER IOERR
C
INUNIT = 50
OPEN(UNIT= INUNIT, FILE='train.dat', STATUS='OLD',
+
IOSTAT = IOERR)
IF (IOERR .NE. 0) THEN
PRINT 210, IOERR
210 FORMAT('can not open training data file, IOERR =',I10)
GOTO 245
ENDIF
C
C
read in the # of test examples

```

```

      READ(INUNIT,*) NUMEX
C      read in the inputs and their corresponding desired outputs
      DO 220 J=1, NUMEX
          READ(INUNIT,*) (INEX(I,J), I=1,INDIM), (OUTEX(I,J), I=1,OUTDIM)
C          READ(INUNIT,*) (OUTEX(I,J), I=1, OUTDIM)
220    CONTINUE
C
      CLOSE (UNIT = INUNIT)
C*****
245    RETURN
      END
C*****

C*****
      SUBROUTINE SETTEX(TESTEX,TEIDX1,TEIDX2,INDIM,NUMTEX)
C*****
C This subroutine is to read the input file, containing the testing
C   examples, and fill out the test example arrays.
C
      INTEGER TEIDX1, TEIDX2, INDIM, NUMTEX
      DOUBLE PRECISION TESTEX(TEIDX1, TEIDX2)
C
      INTEGER IOERR
C
      INUNIT = 50
      OPEN(UNIT= INUNIT, FILE='test.dat', STATUS='OLD',
+        IOSTAT = IOERR)
      IF (IOERR .NE. 0) THEN
          PRINT 260, IOERR
260    FORMAT('can not open test data file, IOERR =',I10)
          GOTO 290
      ENDIF
C
      read in the # of training examples
      READ(INUNIT,*) NUMTEX
C      read in the inputs and their corresponding desired outputs
      DO 270 J=1, NUMTEX
          READ(INUNIT,*) (TESTEX(I,J), I=1,INDIM)
270    CONTINUE
C
      CLOSE (UNIT = INUNIT)
C*****
290    RETURN
      END
C*****

C*****
      SUBROUTINE PRTEX(INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
+        INDIM, OUTDIM, NUMEX)
C*****
C This subroutine is to print out the training examples.
C
      INTEGER IEIDX1,IEIDX2,OEIDX1,OEIDX2,INDIM, OUTDIM, NUMEX
      DOUBLE PRECISION INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2)
C
      PRINT them out
      PRINT 221, NUMEX
      PRINT 222
      DO 230 J=1, NUMEX
          WRITE (*, 223) (INEX(I,J), I=1, INDIM)

```

```

        WRITE (*, 224) (OUTEX(I,J), I=1, OUTDIM)
230  CONTINUE
C
221  FORMAT (/, 'The number of sample is ', I4)
222  FORMAT ('The examples are the following.')
223  FORMAT ('input ', 1X, 30F12.8)
224  FORMAT ('output ', 1X, 30F12.8)
C
C*****
        RETURN
        END
C*****

C*****
SUBROUTINE INITWT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, SEED)
C*****
C This subroutine is to initialize all the connection weights. Each
C weight is initialized to a random number between
C -0.5/Fan-in and 0.5/Fan-in, where Fan-in is the number of nodes
C (including the bias node) in the previous layer.
C The routine will be called after routine SETNET has been called.
C
        INTEGER LAYER(-1:LAIDX1, LAIDX2), WTIDX1
        DOUBLE PRECISION WEIGHT(0:WTIDX1)
C
        INTEGER FANIN, INDEX, SEED
C
        iterate over all layers
        DO 320 K=1, LAYER(-1,1)
C          number of fan-in nodes=number of nodes+1(bias node)
          FANIN=LAYER(K-1,1)+1
C          iterate over all neurons in the layer of target connection
          DO 310 J=1, LAYER(K,1)
C            iterate over all neurons of source connection
            DO 305 I=0, LAYER(K-1, 1)
              INDEX=LAYER(K, 3)+(J-1)*FANIN+I
              WEIGHT(INDEX)=RANDOM(SEED)/FANIN
305          CONTINUE
310          CONTINUE
320          CONTINUE
C
C*****
        RETURN
        END
C*****

C*****
REAL FUNCTION RANDOM(SEED)
C*****
C This random number generator return a random number between -0.5 and
C 0.5.
C Reference: "A PORTABLE RANDOM NUMBER GENERATOR FOR USE IN SIGNAL
C PROCESSING", SANDIA NATIONAL LABORATORIES TECHNICAL
C REPORT, BY S. D. STEARNS.
C
C Input: SEED= an integer
C
        INTEGER SEED
C
        SEED = 2045*SEED + 1
        SEED = SEED - (SEED/1048576)*1048576

```

```

        RANDOM = (SEED+1)/1048577.0 - 0.5
C*****
        RETURN
        END
C*****

C*****
        SUBROUTINE PRTWT(LAYER, LAIDX1, LAIDX2, WEIGHT,WTIDX1)
C*****
C This routine is to print out all connection weights.
C
        INTEGER LAYER(-1:LAIDX1, LAIDX2), WTIDX1
        DOUBLE PRECISION WEIGHT(0:WTIDX1)
        INTEGER INDEX

C
C iterate over all layers
        DO 350 K=1, LAYER(-1,1)
            PRINT 326, K, K-1, LAYER(K-1, 1)
326      FORMAT ('LAYER-', I2, ' (connection from layer ', I2,
+            ' of ', I3, ' nodes)')
C          iterate over all neurons in the layer of target connection
            DO 340 J=1, LAYER(K,1)
                PRINT 328, J
328      FORMAT ('NEURON-', I3)
                INDEX=LAYER(K, 3)+(J-1)*(LAYER(K-1,1)+1)
                WRITE(*,330) (WEIGHT(INDEX+I), I=0,LAYER((K-1),1))
330      FORMAT (100F20.16)
340      CONTINUE
350      CONTINUE
C
C*****
        RETURN
        END
C*****

C*****
        SUBROUTINE NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+            NEURON, NUIDX1, NUIDX2)
C*****
C This routine is to compute the outputs of the network. In the
C process, the output of each neuron is stored in the first row
C of array NEURON(*,1) in a consistent manner. The indexing
C starts at the input layer, then the 1st layer, and, so forth,
C up to the output layer.
C
C Input: inputs NEURON(*,*) with inputs filled at the beginning
C Output: net outputs in NEURON(*), at the end
C
        INTEGER LAIDX1, LAIDX2, WTIDX1, NUIDX1, NUIDX2,
+            LAYER(-1:LAIDX1, LAIDX2)
        DOUBLE PRECISION WEIGHT(0:WTIDX1), NEURON(0:NUIDX1, NUIDX2)
C
        INTEGER FANIN,WINDEX,NINDEX
        DOUBLE PRECISION INNERP, SIG, TEMP, XMAX, XMIN
        COMMON /LIMIT/XMAX, XMIN

C
C forward propagated computation over the layers
        DO 430 K=1, LAYER(-1,1)
C          compute output for each neuron in a layer
            DO 420 J=1, LAYER(K, 1)
C          fan-in of a neuron

```

```

        FANIN=LAYER(K-1,1)+1
C      locate the weight vector of the j-th neuron in Layer K
        WINDEX=LAYER(K,3)+(J-1)*FANIN
C      locate position of j-th neuron in array NEURON
        NINDEX=LAYER(K,2)+J
C      --- fire each neuron
C      compute weighted sum
        TEMP=INNERP(NEURON(LAYER(K-1,2),1),WEIGHT(WINDEX),FANIN)
        IF (TEMP .GE. XMAX) THEN
            NEURON(NINDEX,1)=1.0
        ELSE
            IF (TEMP .LE. XMIN) THEN
                NEURON(NINDEX,1)=0.0
            ELSE
                NEURON(NINDEX,1)=SIG(TEMP)
            ENDIF
        ENDIF
C      compute derivative of the activation function
        NEURON(NINDEX,2)=NEURON(NINDEX,1)*(1.0-NEURON(NINDEX,1))

420     CONTINUE
C      set bias input
        NEURON(LAYER(K,2),1)=-1.0
430     CONTINUE
C*****
        RETURN
        END
C*****

C*****
        DOUBLE PRECISION FUNCTION INNERP(VEC1,VEC2,DIM)
C*****
C This subroutine returns the inner product (or weighted sum) of the
C two vector.
C
        INTEGER DIM
        DOUBLE PRECISION VEC1(DIM), VEC2(DIM)
C
        INNERP=0.0
        DO 440 I=1,DIM
            INNERP=INNERP+VEC1(I)*VEC2(I)
440     CONTINUE
C*****
        RETURN
        END
C*****

C*****
        DOUBLE PRECISION FUNCTION SIG(X)
C*****
C Sigmoid activation function.
C
        DOUBLE PRECISION X
        SIG = 1.0/(1.0+DEXP(-X))
C*****
        RETURN
        END
C*****

C*****
        SUBROUTINE PRTOUT(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1)

```



```

C*****
C This routine is to print out outputs of each neuron in the network.
C
  INTEGER LAIDX1, LAIDX2, NUIDX1
  INTEGER LAYER(-1:LAIDX1, LAIDX2)
  DOUBLE PRECISION NEURON(0:NUIDX1)

C
  PRINT *, 'NETWORK OUTPUT'

C
  DO 470 K=0, LAYER(-1,1)
    WRITE (*,450) K, (NEURON(LAYER(K,2)+J),J=0, LAYER(K, 1))
450    FORMAT ('Layer-',1X,I2,4X, 30F20.16)
470    CONTINUE
C*****
  RETURN
  END
C*****

C*****
  SUBROUTINE COMGRA(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                 NEURON, NUIDX1, NUIDX2, OEXVEC, OUTDIM)
C*****
C This routine is to compute the gradient of the error-squared function.
C The result is stored in the second raw of array WEIGHT ,
C the first raw being the weight vector. This routine requires
C one backward pass through the network. When this routine and the
C routine DEU combined together as one, then only one pass is
C required.
C
  INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2, OUTDIM,
+         LAYER(-1:LAIDX1, LAIDX2)
  DOUBLE PRECISION WEIGHT(0:WTIDX1,WTIDX2), NEURON(0:NUIDX1,NUIDX2),
+         OEXVEC(OUTDIM)

C
  INTEGER FANIN,WINDEX,N1,N2

C
  compute partials of E(w) w.r.t. u(k,j), all k and j.
  CALL DEU(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+         NEURON, NUIDX1, NUIDX2, OEXVEC, OUTDIM)

C
  backward propagated computation over the layers
  DO 530 K=LAYER(-1,1), 1, -1
  C
    over all neurons in output layer
    DO 520 J=1, LAYER(K,1)
  C
    over all fan-in connections of a neuron
    DO 510 I=0, LAYER(K-1,1)
  C
    fan-in of a neuron
    FANIN=LAYER(K-1,1)+1
  C
    locate position in weight vector
    WINDEX=LAYER(K,3)+(J-1)*FANIN+I
  C
    locate positions in array NEURON
    N1=LAYER(K,2)+J
    N2=LAYER(K-1,2)+I
  C
    compute partial of E(W) w.r.t. w(i,j,k)
    WEIGHT(WINDEX,2)= NEURON(N1,2)*NEURON(N2,1)*NEURON(N1,3)
510    CONTINUE
520    CONTINUE
530    CONTINUE
C*****
  RETURN

```

```

      END
C*****

C*****
      SUBROUTINE DEU(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                 NEURON, NUIDX1, NUIDX2, OEXVEC, OUTDIM)
C*****
C This routine is to compute the partial derivatives of the error-
C squared function w.r.t the output of each neuron. The result
C is stored in the 3rd row of array NEURON. This computation
C requires one backward pass through the network.
C The partial derivatives are needed in the computation of the
C gradient and the Hessian matrix of E(w). If only the gradient
C is needed, then this procedure could be integrated into the
C subroutine COMGRA, computing the gradient of E(w).
C
      INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2, OUTDIM,
+           LAYER(-1:LAIDX1, LAIDX2)
      DOUBLE PRECISION WEIGHT(0:WTIDX1,WTIDX2), NEURON(0:NUIDX1,NUIDX2),
+           OEXVEC(OUTDIM)

      INTEGER WINDEX, NINDEX, N1, S
      DOUBLE PRECISION TEMP1

C
C --- backward propagated computation over the layers
C base case when K=output layer
      K=LAYER(-1,1)
C over all neurons in output layer
      DO 610 J=1, LAYER(K,1)
C     locate position in array NEURON
      NINDEX=LAYER(K,2)+J
      NEURON(NINDEX,3)=NEURON(NINDEX,1)-OEXVEC(J)
610  CONTINUE
C
C backpropagation over all other layers
      DO 640 K=LAYER(-1,1)-1, 1,-1
C     over all neurons in a layer
      DO 630 J=1, LAYER(K,1)
C     locate positions in array NEURON for storage
      NINDEX=LAYER(K,2)+J
      NEURON(NINDEX,3)=0
C     over all neurons in the next layer
      DO 620 S=1, LAYER(K+1,1)
C     locate position in weight vector
      WINDEX=LAYER(K+1,3)+(S-1)*(LAYER(K,1)+1)+J
C     locate positions in array NEURON
      N1=LAYER(K+1,2)+S
C     compute partial of E(W) w.r.t. u(k,j)
      TEMP1=NEURON(N1,2)*WEIGHT(WINDEX,1)*NEURON(N1,3)
      NEURON(NINDEX,3)=NEURON(NINDEX,3)+TEMP1
620  CONTINUE
630  CONTINUE
640  CONTINUE
C*****
      RETURN
      END
C*****

C*****
      SUBROUTINE STEDES(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+                 NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,

```

```

+           OUTEX, OEIDX1, OEIDX2, NUMEX, WTDIM,
+           INDIM, OUTDIM, STPSIZ, TOLERA)
C*****
C This routine implements the Steepest Descent learning algorithm 3.4.1
C   in batch model.
C
C   INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2, WTDIM,
+     IEIDX1, IEIDX2, OEIDX1, OEIDX2, NUMEX, INDIM, OUTDIM,
+     LAYER(-1:LAIDX1, LAIDX2)
C   DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2), NEURON(0:NUIDX1, NUIDX2),
+     INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2),
+     STPSIZ, TOLERA
C
C   INTEGER P, EPOCH
C   DOUBLE PRECISION RMS, RMSBAK, RR, EA, EB, EC, A, B, C, XTOL
C
C   EPOCH=0
C --- get initial RMS
C   RMSBAK=0.0
C   DO 1006 P=1, NUMEX
C     setup inputs
C     DO 1007 I=1, INDIM
C       NEURON(I,1)=INEX(I,P)
1007   CONTINUE
C     NEURON(0,1)=-1.0
C     compute outputs of all neurons in the network
C     CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+              NEURON, NUIDX1, NUIDX2)
C     accumulate error over all patterns
C     RMSBAK=RMSBAK+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+              OUTEX(1,P), OUTDIM)
1006   CONTINUE
C     RMSBAK=DSQRT(RMSBAK/NUMEX)
C     WRITE (*,1062) RMSBAK
C
1001   DO 1005 I=0, WTDIM-1
C     WEIGHT(I,3)=0.0
1005   CONTINUE
C
C --- Batch model: accumulate error information over all patterns
C   DO 1030 P=1, NUMEX
C     setup inputs
C     DO 1010 I=1, INDIM
C       NEURON(I,1)=INEX(I,P)
1010   CONTINUE
C     NEURON(0,1)=-1.0
C
C     compute outputs of all neurons in the network
C     CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+              NEURON, NUIDX1, NUIDX2)
C
C     compute the gradient of E(w) for one pattern
C     CALL COMGRA(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+              NEURON, NUIDX1, NUIDX2, OUTEX(1,P), OUTDIM)
C     accumulate the NEGATIVE partials over all patterns
C     DO 1020 I=0, WTDIM-1
C       WEIGHT(I,3)=WEIGHT(I,3)-WEIGHT(I,2)
1020   CONTINUE
1030   CONTINUE
C
C   find a line search interval

```

```

A=0.0D0
CALL FIDITV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,WTIDX2,
+          WTDIM, NEURON, NUIDX1, NUIDX2, NUMEX,
+          INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
+          INDIM, OUTDIM, EA, EB, EC, A, B, C, STPSIZ)
IF (A.EQ.0.0) THEN
  XTOL=0.5
  CALL QINTER(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,WTIDX2,
+          WTDIM, NEURON, NUIDX1, NUIDX2, NUMEX,
+          INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
+          INDIM, OUTDIM, EA, EB, EC, A, B, C,XTOL,STPSIZ)
ELSE IF (A.EQ.-1.0) THEN
  PRINT *, 'SEARCH FAILED'
  GOTO 1070
ELSE
  PRINT *, 'Very big step encountered with B= ', B
ENDIF
C
C use the fixed step size to update the weights
DO 1040 I=0, WTDIM-1
  WEIGHT(I,1)=WEIGHT(I,1)+WEIGHT(I,3)*C*STPSIZ
1040 CONTINUE
C
  EPOCH=EPOCH+1
  RMS=DSQRT(EC*2.0/NUMEX)
c IF (MOD(EPOCH, 100) .EQ. 0) THEN
  WRITE (*,1062) RMS, EPOCH, C
1062  FORMAT('RMS error= ',F23.19,'EPOCH= ',I6,'step size= ',G15.7)
c ENDIF
C check if the convergence criterion is satisfied.
IF (RMSBAK-RMS .GE. TOLERA) THEN
  RMSBAK=RMS
  GOTO 1001
ENDIF
C otherwise, training done
C
C*****
1070 RETURN
END
C*****

C*****
DOUBLE PRECISION FUNCTION RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1,
+          NUIDX2, OEXVEC, OUTDIM)
C*****
C This routine is to compute the errors of the network outputs
C against the desired outputs.
C
  INTEGER LAIDX1, LAIDX2, NUIDX1, NUIDX2, OUTDIM
  INTEGER LAYER(-1:LAIDX1, LAIDX2)
  DOUBLE PRECISION NEURON(0:NUIDX1, NUIDX2), OEXVEC(OUTDIM)
C
  INTEGER NINDEX, K
C
  RR=0.0
  K=LAYER(-1,1)
C over all neurons in output layer
DO 1110 J=1, LAYER(K,1)
C locate position in array NEURON
  NINDEX=LAYER(K,2)+J
  RR=RR+(NEURON(NINDEX,1)-OEXVEC(J))**2

```

```

1110 CONTINUE
C
C*****
      RETURN
      END
C*****
C
C*****
      SUBROUTINE FIDITV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,WTIDX2,
+                    WTDIM, NEURON, NUIDX1, NUIDX2, NUMEX,
+                    INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
+                    INDIM, OUTDIM, EA, EB, EC, A, B, C, STEP)
C*****
C This routine is to find an interval (a, b) and a point c on (a, b)
C in order to perform a quadratic interpolation for a proper choice
C of the weight-update step size. Note that a=0.0 always.
C First, the program will try to locate an interval (0, x) and the
C point c satisfying the conditions for a quadratic interpolation.
C If the attempt failed, the routine will either reach out beyond
C 100.0 for a possible maximum step size that still results in
C reduction of function values, in which case no interpolation will
C be performed, or fail to locate an interval.
C On exit:
C   A=0.0 ==> interval found (then, a quadratic interpolation is
C             performed).
C   A=-1.0 ==> interval not found, method failed.
C   A=-2.0 ==> interval not found, use C as step size.
C
      INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2, NUMEX,
+          LAYER(-1:LAIDX1, LAIDX2), IEIDX1, IEIDX2, OEIDX1, OEIDX2,
+          INDIM, OUTDIM, WTDIM
      DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2),
+          INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2),
+          NEURON(0:NUIDX1, NUIDX2),
+          EA, EB, EC, A, B, C, STEP
      COMMON /LIMIT1/LIM1, LIM2, LIM3
C
      INTEGER FANIN, WINDEX, NINDEX, P
      DOUBLE PRECISION INNERP, SIG, TEMP, RR
C
C --- get E(A) with a step size A=0, i.e., at w=w+0.0*step*dw
      A=0.0
      EA=0.0
      DO 1332 P=1, NUMEX
C          setup inputs
          DO 1331 I=1, INDIM
              NEURON(I,1)=INEX(I,P)
1331          CONTINUE
              NEURON(0,1)=-1.0
C
          compute outputs of all neurons in the network
          CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+                   NEURON, NUIDX1, NUIDX2)
C          accumulate error over all patterns
          EA=EA+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+               OUTEX(1,P), OUTDIM)
1332          CONTINUE
          EA=0.5*EA
C
C --- find the internal point C beginning at 2.0, then as close to A=0.0
C as possible such that E(c) < E(a).

```

```

C
  C=4.0
  DO 1342 J=1, LIM1
    C=C*0.5
C    update weights with step size C=1.0
    DO 1340 I=0, WTDIM-1
      WEIGHT(I,4)=WEIGHT(I,1)+WEIGHT(I,3)*STEP*C
1340    CONTINUE
    EC=0.0
    DO 1334 P=1, NUMEX
C      setup inputs
      DO 1333 I=1, INDIM
        NEURON(I,1)=INEX(I,P)
1333    CONTINUE
      NEURON(0,1)=-1.0
C
C      compute outputs of all neurons in the network
      CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT(0,4), WTIDX1,
+        NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
      EC=EC+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+        OUTEX(1,P), OUTDIM)
1334    CONTINUE
    EC=0.5*EC
    IF (EC.LT.EA) GOTO 1343
1342  CONTINUE
C  otherwise, search failed, exit with error code A= -1.0
  A=-1.0
  GOTO 1390

C --- locate the right end point B
1343  B=C+1.0
      DO 1370, J=1, LIM2
        IF (J.GT.LIM3) THEN
C          take bigger step
          B=B+10.0
        ELSE
          B=B+1.0
        ENDIF
      EB=0.0
      DO 1361 I=0, WTDIM-1
        WEIGHT(I,4)=WEIGHT(I,1)+WEIGHT(I,3)*STEP*B
1361    CONTINUE
      DO 1363 P=1, NUMEX
C      setup inputs
      DO 1362 I=1, INDIM
        NEURON(I,1)=INEX(I,P)
1362    CONTINUE
      NEURON(0,1)=-1.0
C
C      compute outputs of all neurons in the network
      CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT(0,4), WTIDX1,
+        NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
      EB=EB+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+        OUTEX(1,P), OUTDIM)
1363    CONTINUE
    EB=0.5*EB
C  terminate if new E(b) > E(c)
    IF (EB .GT. EC) GOTO 1390
    IF (EB .LT. EC) THEN

```

```

                EC=EB
                C=B
            ENDIF
1370 CONTINUE
C
C    otherwise, search failed, exit with error code A= -2.0
    C=B
    EC=EB
    A=-2.0

C*****
1390 RETURN
    END
C*****
C
C*****
SUBROUTINE QINTER(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+               WTDIM, NEURON, NUIDX1, NUIDX2, NUMEX,
+               INEX, IEIDX1, IEIDX2, OUTEX, OEIDX1, OEIDX2,
+               INDIM, OUTDIM, EA, EB, EC, A, B, C, XTOL, STEP)
C*****
C This routine performs a quadratic interpolation on the points
C (a, E(a)), (b, E(b)), and (c, E(c)) over the interval (a, b).
C Successive iterations of quadratic interpolations are performed
C until the interval (a, b) is reduced within tolerance. On exit,
C point c is the desired step size.
C
    INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2, NUMEX,
+         LAYER(-1:LAIDX1, LAIDX2), IEIDX1, IEIDX2, OEIDX1, OEIDX2,
+         INDIM, OUTDIM, WTDIM
    DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2),
+         NEURON(0:NUIDX1, NUIDX2),
+         INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2),
+         EA, EB, EC, A, B, C, EX, X, XTOL, STEP
    COMMON /LIMIT2/LIM4

C
    INTEGER CONTER, P
    DOUBLE PRECISION INNERP, SIG, TEMP, RR, Y, Z

C
    CONTER=0
C compute point of interpolation
1405 Y= ((B**2-C**2)*EA+(C**2-A**2)*EB+(A**2-B**2)*EC)
    Z= (2.0*((B-C)*EA+(C-A)*EB+(A-B)*EC))
    IF (Z .EQ. 0.0) GOTO 1440
    X=Y/Z

C
C compute net output at w+x*dw
DO 1410 I=0, WTDIM-1
    WEIGHT(I,4)=WEIGHT(I,1)+X*STEP*WEIGHT(I,3)
1410 CONTINUE
C compute squared-error at w+X*dw
EX=0.0
DO 1430 P=1, NUMEX
C setup inputs
DO 1420 I=1, INDIM
    NEURON(I,1)=INEX(I,P)
1420 CONTINUE
    NEURON(0,1)=-1.0

C
C compute outputs of all neurons in the network
CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT(0,4), WTIDX1,

```

```

+          NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
      EX=EX+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+          OUTEX(1, P), OUTDIM)
1430 CONTINUE
      EX=0.5*EX
C
      IF ( X.LT.C .AND. EX.LT.EC ) THEN
          B=C
          C=X
          EB=EC
          EC=EX
      ELSE IF ( X.GT.C .AND. EX.GT.EC ) THEN
          B=X
          EB=EX
      ELSE IF ( X.LT.C .AND. EX.GT.EC ) THEN
          A=X
          EA=EX
      ELSE
          A=C
          C=X
          EA=EC
          EC=EX
      ENDIF
C
      CONTER=CONTER+1
C      check stopping condition
      IF (CONTER.EQ.LIM4) GOTO 1440
      IF (B-C.LT.XTOL .OR. C-A.LT.XTOL) GOTO 1440
      GOTO 1405

C*****
1440 RETURN
      END
C*****
C
C
C*****
      SUBROUTINE DUUVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+          NEURON, NUIDX1, NUIDX2, UUVV, UUIDX1, UUIDX2, UUIDX3)
C*****
C This routine is to compute the partial derivatives of each neuron
C output function w.r.t the output of each neuron in its connection
C path and, for convenience, the partial derivatives of the weighted
C sum function of each neuron w.r.t a previous weighted sum in its
C input path. The result of the former is stored in the first
C raw of array UUVV and the latter in the 2nd raw. This
C computation requires one forward pass through the network.
C The partial derivatives are needed in the computation of the
C Jacobian and the Hessian matrices of E(w).
C
      INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2,
+          LAYER(-1:LAIDX1, LAIDX2), UUIDX1, UUIDX2, UUIDX3
      DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2), NEURON(0:NUIDX1, NUIDX2),
+          UUVV(0:UUIDX1, 0:UUIDX2, UUIDX3)

      INTEGER WINDEX, N1, N2, N3, S
      DOUBLE PRECISION TEMP1
C
C forward propagated computation over the layers
      DO 2050 K2=1, LAYER(-1,1)

```



```

C      over all neurons in a layer
      DO 2040 I=1, LAYER(K2,1)
      N2=LAYER(K2,2)+I
C      for all neuron in later layers
      DO 2030 K1=K2, LAYER(-1,1)
      DO 2020 J=1, LAYER(K1,1)
      N1=LAYER(K1,2)+J
      IF (K2 .EQ. K1) THEN
      IF (J .EQ. I) THEN
      UUVV(N1, N2,1)=1
      UUVV(N1, N2,2)=1
      ELSE
      UUVV(N1, N2,1)=0
      UUVV(N1, N2,2)=0
      ENDIF
      ELSE IF (K1 .EQ. K2+1) THEN
      WINDEX=LAYER(K1,3)+(J-1)*(LAYER(K1-1,1)+1)+I
      UUVV(N1, N2,1)=NEURON(N1,2)*WEIGHT(WINDEX,1)
      UUVV(N1, N2,2)=NEURON(N2,2)*WEIGHT(WINDEX,1)
      ELSE
      K=K1-1
C      over all neurons in previous layer
      TEMP1=0.0
      DO 2010 S=1, LAYER(K, 1)
C
      WINDEX=LAYER(K1,3)+(J-1)*(LAYER(K,1)+1)+S
      N3=LAYER(K,2)+S
      TEMP1=TEMP1+WEIGHT(WINDEX,1)*UUVV(N3,N2,1)
2010      CONTINUE
      UUVV(N1,N2,1)=TEMP1*NEURON(N1,2)
      UUVV(N1,N2,2)=TEMP1*NEURON(N2,2)
      ENDIF
2020      CONTINUE
2030      CONTINUE
2040      CONTINUE
2050      CONTINUE
C
C*****
      RETURN
      END
C*****

C*****
SUBROUTINE COMJAC(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+
+
+
      NEURON, NUIDX1, NUIDX2, HESIAN, WTDIM,
      UUVV, UUIDX1, UUIDX2, UUIDX3, OUTDIM)
C*****
C This routine is to compute the Jacobian matrix of the net function and
C approximate the Hessian matrix by using the Jacobian matrix.
C The 1st row of the J-matrix is over all the weights and the 2nd
C over output dimension. The matrix is actually stored in array
C WEIGHT, starting at the 5th row of array WEIGHT. The estimated
C Hessian matrix is stored in matrix HESIAN.
C
C
C
      INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2, WTDIM,
+
+
+
      OUTDIM, UUIDX1, UUIDX2, UUIDX3, LAYER(-1:LAIDX1, LAIDX2)
      DOUBLE PRECISION WEIGHT(0:WTIDX1,WTIDX2),NEURON(0:NUIDX1,NUIDX2),
+
+
+
      UUVV(0:UIDX1, 0:UIDX2, UUIDX3),
      HESIAN(0:WTIDX1, 0:WTIDX1)
C

```

```

      INTEGER WINDEX,N1,N2,N3, M
C
C   over output functions of all neurons in the last layer
DO 2250, M=1, OUTDIM
C   over all the connection weights
C   over layers
DO 2240 K=1, LAYER(-1,1)
C   over all neurons in a layer
DO 2230 J=1, LAYER(K,1)
C   over all neurons in the previous layer
DO 2220 I=0, LAYER(K-1,1)
      N1=LAYER(LAYER(-1,1),2)+M
      N2=LAYER(K,2)+J
      N3=LAYER(K-1,2)+I
      WINDEX=LAYER(K,3)+(J-1)*(LAYER(K-1,1)+1)+I
      WEIGHT(WINDEX,M+4)= UUVV(N1,N2,1)*NEURON(N2,2)
+
      *NEURON(N3,1)
2220      CONTINUE
2230      CONTINUE
2240      CONTINUE
2250      CONTINUE
C
C   the following estimate the Hessian matrix of E(w)
C
DO 2280 J=0, WTDIM-1
  DO 2270 I=J, WTDIM-1
    HESIAN(I,J)=0.0
    DO 2260 M=1, OUTDIM
      HESIAN(I,J)=WEIGHT(J,M+4)*WEIGHT(I,M+4)+HESIAN(I,J)
2260      CONTINUE
      HESIAN(J,I)=HESIAN(I,J)
2270      CONTINUE
2280      CONTINUE
C
C*****
2299      RETURN
      END
C*****

```

```

C*****
      SUBROUTINE LSOLV (A,BX,N,LDIM,NRANK,PSMAL)
C
C   LSOLV 1.6          APRIL 1992
C
C   J. P. CHANDLER, COMPUTER SCIENCE DEPARTMENT,
C   OKLAHOMA STATE UNIVERSITY
C
C   LSOLV SOLVES A SYSTEM OF LINEAR EQUATIONS USING GAUSSIAN
C   ELIMINATION WITH PARTIAL PIVOTING.
C   IF THE MATRIX OF COEFFICIENTS IS SINGULAR, LSOLV COMPUTES
C   THE SOLUTION THAT WOULD RESULT FROM MULTIPLYING A RAO
C   PSEUDOINVERSE OF THE COEFFICIENT MATRIX TIMES THE VECTOR OF
C   CONSTANTS.
C   C. R. RAO AND S. K. MITRA, -GENERALIZED INVERSE OF MATRICES
C   AND ITS APPLICATIONS- (WILEY, 1971), PAGE 212
C
C   N IS THE NUMBER OF EQUATIONS IN THE LINEAR SYSTEM.
C   ON ENTRY, A(*,*) CONTAINS THE MATRIX OF COEFFICIENTS AND
C   BX(*) CONTAINS THE VECTOR OF CONSTANTS (THE RIGHTHAND
C   SIDES).

```

```

C ON EXIT, BX(*) CONTAINS THE SOLUTION VECTOR AND A(*,*)
C CONTAINS GARBAGE.
C LDIM IS THE VALUE OF THE DIMENSIONS OF THE ARRAYS A AND BX.
C THE VALUE OF N MUST NOT EXCEED THE VALUE OF LDIM.
C NRANK RETURNS THE RANK OF THE MATRIX A.
C IF NRANK .LT. N THEN THE MATRIX A WAS SINGULAR.
C PSMAL RETURNS THE PIVOT, IF ANY, THAT HAD THE SMALLEST
C NONZERO MAGNITUDE.
C*****
C
C
C      DOUBLE PRECISION A, BX, ZABS, ARG, BIGA, TEMP, EM, SUM, PSMAL,
*      RZERO
C
C      DIMENSION A(LDIM,N), BX(N)
C
C      ZABS(ARG)=DABS(ARG)
C
C CHECK FOR AN INVALID VALUE OF N OR LDIM.
C
C      RZERO=0.0D0
C      NRANK=-1
C      PSMAL=RZERO
C      IF(N.LT.1 .OR. N.GT.LDIM) GO TO 2495
C
C TRIANGULARIZE THE MATRIX A.
C
C      NRANK=0
C      IF(N.LT.2) GO TO 2470
C      NMU=N-1
C      DO 2460 J=1, NMU
C
C SEARCH COLUMN J FOR THE PIVOT ELEMENT.
C
C      BIGA=RZERO
C      DO 2410 K=J, N
C          TEMP=ZABS(A(K,J))
C          IF(TEMP.LE.BIGA) GO TO 2410
C          BIGA=TEMP
C          JPIV=K
2410      CONTINUE
C          IF(BIGA.LE.RZERO) GO TO 2460
C          IF(JPIV.LE.J) GO TO 2430
C
C INTERCHANGE EQUATIONS J AND JPIV.
C
C      DO 2420 L=J, N
C          TEMP=A(J,L)
C          A(J,L)=A(JPIV,L)
C          A(JPIV,L)=TEMP
2420      CONTINUE
C          TEMP=BX(J)
C          BX(J)=BX(JPIV)
C          BX(JPIV)=TEMP
2430      JPU=J+1
C          DO 2450 K=JPU, N
C
C PERFORM ELIMINATION ON EQUATION K.
C
C          EM=A(K,J)/A(J,J)
C          IF(EM.EQ.RZERO) GO TO 2450

```

```

                DO 2440 L=JPU,N
                  A(K,L)=A(K,L)-EM*A(J,L)
2440             CONTINUE
                  BX(K)=BX(K)-EM*BX(J)
2450             CONTINUE
2460             CONTINUE
C
C DO THE BACK SOLUTION.
C
2470 DO 2490 JINV=1,N
              J=N+1-JINV
              TEMP=A(J,J)
              IF(TEMP.NE.RZERO) GO TO 2475
              BX(J)=RZERO
              GO TO 2490
2475         NRANK=NRANK+1
              IF(PSMAL.EQ.RZERO .OR. ZABS(TEMP).LT.ZABS(PSMAL))
*             PSMAL=TEMP
              SUM=RZERO
              IF(J.GE.N) GO TO 2485
              JPU=J+1
              DO 2480 K=JPU,N
                SUM=SUM+A(J,K)*BX(K)
2480             CONTINUE
C
2485         BX(J)=(BX(J)-SUM)/TEMP
2490         CONTINUE
C
2495         RETURN
C
C END LSOLV
C
                END

```

```

C*****
SUBROUTINE GNLM(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+             NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+             OUTEX, OEIDX1, OEIDX2, NUMEX,
+             INDIM, OUTDIM, TOLERA,
+             UUVV, UUIDX1, UUIDX2, UUIDX3,
+             HESIAN, WTDIM)
C*****
C This routine implements the Gauss-Newton/Levenberg-Marquardt learning
C algorithm, with an add-in identity matrix I.
C
INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2,
+       IEIDX1, IEIDX2, OEIDX1, OEIDX2, NUMEX, INDIM, OUTDIM, WTDIM,
+       LAYER(-1:LAIDX1, LAIDX2), UUIDX1, UUIDX2, UUIDX3
DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2), NEURON(0:NUIDX1, NUIDX2),
+       INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2),
+       TOLERA, UUVV(0:UUIX1, 0:UUIX2, UUIX3),
+       HESIAN(0:WTIDX1, 0:WTIDX1, 3)
C
INTEGER P, EPOCH, NRANK, DIM, LDIM
DOUBLE PRECISION RMS, RMSBAK, RR, LAMBDA, FACTOR, PSMAI, NORM,
+       INILAM, INIFAC, CUTOFF, SMALL
COMMON /LEVMAI/INILAM, INIFAC, CUTOFF, SMALL
C
PRINT *, 'Gauss-Newton/Levenberg-Marquardt(I) is working'

```

```

PRINT *, 'tolerance= ', TOLERA
C
EPOCH=0
LAMBDA=INILAM
FACTOR=INIFAC
PRINT *, 'FACTOR= ', FACTOR
C --- get initial RMS
RMSBAK=0.0
DO 2506 P=1, NUMEX
C     setup inputs
DO 2507 I=1, INDIM
    NEURON(I,1)=INEX(I,P)
2507 CONTINUE
    NEURON(0,1)=-1.0
C     compute outputs of all neurons in the network
CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+          NEURON, NUIDX1, NUIDX2)
C     accumulate error over all patterns
RMSBAK=RMSBAK+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+          OUTEX(1,P), OUTDIM)
2506 CONTINUE
C     PRINT*, 'accumulated RMSBAK= ', RMSBAK
RMSBAK=DSQRT(RMSBAK/NUMEX)
WRITE (*,2562) RMSBAK, EPOCH, LAMBDA, FACTOR
C
2501 DO 2505 J=0, WTDIM-1
    DO 2502 I=J, WTDIM-1
        HESIAN(I,J,3)=0.0
        HESIAN(J,I,3)=0.0
2502 CONTINUE
    WEIGHT(J,3)=0.0
2505 CONTINUE
C
C --- Batch model: accumulate error information over all patterns
DO 2540 P=1, NUMEX
C     setup inputs
DO 2510 I=1, INDIM
    NEURON(I,1)=INEX(I,P)
2510 CONTINUE
    NEURON(0,1)=-1.0
C
C     compute outputs of all neurons in the network
CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+          NEURON, NUIDX1, NUIDX2)
C
C     compute the gradient of E(w) for one pattern
CALL COMGRA(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+          NEURON, NUIDX1, NUIDX2, OUTEX(1,P), OUTDIM)
C     accumulate the NEGATIVE gradient over all patterns
DO 2520 I=0, WTDIM-1
    WEIGHT(I,3)=WEIGHT(I,3) - WEIGHT(I,2)
2520 CONTINUE
C
C     compute the useful partials
CALL DUUVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+          NEURON, NUIDX1, NUIDX2, UUVV, UUIDX1, UUIDX2, UUIDX3)
C     compute the Jacobian matrix and estimate the Hessian
CALL COMJAC(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+          NEURON, NUIDX1, NUIDX2, HESIAN, WTDIM,
+          UUVV, UUIDX1, UUIDX2, UUIDX3, OUTDIM)
C

```

```

C      sum up the Hessians over all patterns
      DO 2535 J=0, WTDIM-1
        DO 2530 I=J, WTDIM-1
          HESIAN(I,J,3)=HESIAN(I,J,3) + HESIAN(I,J,1)
          HESIAN(J,I,3)=HESIAN(I,J,3)
2530      CONTINUE
2535      CONTINUE
2540      CONTINUE
C
C --- formulate the linear system equations
      IF ( LAMBDA .LT. CUTOFF ) THEN
        LAMBDA=LAMBDA/10
      ELSE
        LAMBDA=LAMBDA/FACTOR
      ENDIF
C
C      add in Levenberg-Marquardt parameters
2545      DO 2555 J=0, WTDIM-1
        DO 2550 I=J, WTDIM-1
          HESIAN(I,J,2)=HESIAN(I,J,3)
          HESIAN(J,I,2)=HESIAN(I,J,3)
2550      CONTINUE
        WEIGHT(J,2)=WEIGHT(J,3)
        HESIAN(J,J,2)=HESIAN(J,J,3) + LAMBDA
2555      CONTINUE
C --- setup parameters for calling LSOLVE
2557      FORMAT(20F20.10)
2561      DIM = WTDIM
        LDIM=WTIDX1+1
        CALL LSOLV(HESIAN(0,0,2),WEIGHT(0,2),DIM,LDIM,NRANK,PSMAL)
C      checking singularity
      IF (NRANK .LT. WTDIM) THEN
        PRINT*, 'singular system'
        LAMBDA=LAMBDA*FACTOR
        DO 2563 J=0, WTDIM-1
          DO 2564 I=J, WTDIM-1
            HESIAN(I,J,2)=HESIAN(I,J,3)
            HESIAN(J,I,2)=HESIAN(I,J,3)
2564          CONTINUE
          HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA
2563        CONTINUE
        GOTO 2561
      ENDIF
C
C      temporary updating weights
      NORM=0.0
      DO 2660, I=0, WTDIM-1
        NORM=NORM+WEIGHT(I,2)**2
        WEIGHT(I,2)=WEIGHT(I,1) + WEIGHT(I,2)
2660      CONTINUE
      NORM=DSQRT(NORM)
C
C      check learning result
      RMS=0.0
      DO 2570 P=1, NUMEX
C      setup inputs
        DO 2575 I=1, INDIM
          NEURON(I,1)=INEX(I,P)
2575        CONTINUE
        NEURON(0,1)=-1.0
C      compute outputs of all neurons in the network

```

```

        CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT(0,2), WTIDX1,
+
+           NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
        RMS=RMS+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+
+           OUTEX(1,P), OUTDIM)
C      PRINT*, 'accumulating RMS=', RMS
2570  CONTINUE
C
C      PRINT *, 'accumulated RMS= ', RMS
        RMS=DSQRT(RMS/NUMEX)
        WRITE (*,2562) RMS, EPOCH, LAMBDA, NORM
2562  FORMAT('RMS=', F23.19, I10, ' LAMBDA=', 2G15.7)
        IF (RMS .GT. RMSBAK ) THEN
            IF (LAMBDA .LT. CUTOFF) THEN
                LAMBDA=CUTOFF
            ELSE
                LAMBDA=LAMBDA*FACTOR
            ENDIF
            GOTO 2545
        ENDIF

C      EPOCH=EPOCH+1
C      update weights
        DO 2580 I=0, WTDIM-1
            WEIGHT(I,1)=WEIGHT(I,2)
2580  CONTINUE
C      check if the convergence criterion is satisfied.
        IF (RMSBAK-RMS .GE. TOLERA) THEN
            RMSBAK=RMS
            GOTO 2501
        ENDIF
C      otherwise, training done
C
C*****
        RETURN
        END
C*****

C*****
SUBROUTINE DEVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+
+           NEURON, NUIDX1, NUIDX2,
+
+           UUVV, UUIDX1, UUIDX2, UUIDX3)
C*****
C This routine is to compute the 2nd-order derivatives of the error-
C squared function w.r.t. the weighted sum functions v(k1,j1) and
C v(k2,j2), with k1>=k2. The result is stored in UUVV(*,*,3).
C
C
C      INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2,
+
+           UUIDX1, UUIDX2, UUIDX3, LAYER(-1:LAIDX1, LAIDX2)
        DOUBLE PRECISION WEIGHT(0:WTIDX1,WTIDX2), NEURON(0:NUIDX1,NUIDX2),
+
+           UUVV(0:UIDX1, 0:UIDX2, UUIDX3)
C
C      INTEGER WINDEX, N1, N2, N3, S
        DOUBLE PRECISION TEMP1, TEMP2
C
C      backward propagation over the net
        DO 2180 K2=1, LAYER(-1,1)
C      over all neurons in a layer
        DO 2170 J2=1, LAYER(K2,1)
            N2=LAYER(K2,2)+J2

```

```

C          base case when k1=K, the output layer
DO 2110 J1=1, LAYER(LAYER(-1,1),1)
      N1=LAYER(LAYER(-1,1),2)+J1
      UUVV(N1,N2,3)= UUVV(N1,N2,2)*( NEURON(N1,2)**2 +
+
+          NEURON(N1,3)*NEURON(N1,2) *
          (1.0 - 2.0*NEURON(N1,1)) )
2110      CONTINUE
C
C          backward propagation until layer k2
DO 2160 K1=LAYER(-1,1)-1, K2, -1
      DO 2150 J1=1, LAYER(K1,1)
        N1=LAYER(K1,2)+J1
        TEMP1=0.0
        TEMP2=0.0
C          over all neurons in the next layer
DO 2130 S=1, LAYER(K1+1,1)
      WINDEX=LAYER(K1+1,3) + (S-1)*(LAYER(K1,1)+1) + J1
      N3=LAYER(K1+1,2)+S
      TEMP1=TEMP1 + WEIGHT(WINDEX,1)*NEURON(N3,2)*
+
+          NEURON(N3,3)
      TEMP2=TEMP2 + WEIGHT(WINDEX,1)*UUVV(N3,N2,3)
2130      CONTINUE
      UUVV(N1,N2,3)= TEMP1*NEURON(N1,2)*
+
+          (1.0-2.0*NEURON(N1,1))*UUVV(N1,N2,2)
+          + TEMP2*NEURON(N1,2)
C
2150          CONTINUE
2160          CONTINUE
2170          CONTINUE
2180          CONTINUE
C
C*****
      RETURN
      END
C*****

C*****
      SUBROUTINE COMHAS(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+
+          UUVV, UUIDX1, UUIDX2, UUIDX3, HESIAN, WTIDX1)
C*****
C This routine is to compute the Hessian matrix of the error function.
C The result is stored in matrix HESIAN(*,*,1).
C
C
C
      INTEGER LAIDX1, LAIDX2, NUIDX1, NUIDX2, WTIDX1,
+
+          UUIDX1, UUIDX2, UUIDX3, LAYER(-1:LAIDX1, LAIDX2)
      DOUBLE PRECISION NEURON(0:NUIDX1,NUIDX2),
+
+          UUVV(0:UIDX1, 0:UIDX2, UUIDX3),
+
+          HESIAN(0:WTIDX1, 0:WTIDX1)
C
      INTEGER W1,W2,N1,N2,N3,N4,I,J,I1,I2,J1,J2,K1,K2
C
DO 2380 K2=1, LAYER(-1,1)
C over all neurons a layer
DO 2370 J2=1, LAYER(K2,1)
      N2=LAYER(K2,2)+J2
C over all neurons in the previous layer
DO 2360 I2=0, LAYER(K2-1,1)
      N4=LAYER(K2-1,2)+I2
      W2=LAYER(K2,3)+(J2-1)*(LAYER(K2-1,1)+1)+I2
C for all neuron in later layers

```



```

DO 2350 K1=K2, LAYER(-1,1)
  IF (K1.EQ.K2) THEN
    I=I2
    J=J2
  ELSE
    I=0
    J=1
  ENDIF
DO 2340 J1=J, LAYER(K1,1)
  N1=LAYER(K1,2)+J1
  DO 2330 I1=I, LAYER(K1-1,1)
    N3=LAYER(K1-1,2)+I1
    W1=LAYER(K1,3)+(J1-1)*(LAYER(K1-1,1)+1)+I1
    IF (K1.EQ.K2) THEN
      HESIAN(W1,W2)=
+       UUVV(N1,N2,3)*NEURON(N4,1)*NEURON(N3,1)
    ELSE
      HESIAN(W1,W2)=
+       UUVV(N1,N2,3)*NEURON(N4,1)*NEURON(N3,1)+
+       NEURON(N1,3)*NEURON(N1,2)*NEURON(N3,2)*
+       UUVV(N3,N2,2)*NEURON(N4,1)
    ENDIF
    HESIAN(W2,W1)=HESIAN(W1,W2)
2330     CONTINUE
2340     CONTINUE
2350     CONTINUE
2360     CONTINUE
2370     CONTINUE
2380 CONTINUE
C
C
C*****
      RETURN
      END
C*****

C*****
      SUBROUTINE NLM(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+       NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+       OUTEX, OEIDX1, OEIDX2, NUMEX,
+       INDIM, OUTDIM, TOLERA,
+       UUVV, UUIDX1, UUIDX2, UUIDX3,
+       HESIAN, WTDIM)
C*****
C This routine implements the Damped Newton learning algorithm
C (Algorithm 4.2.1), adding a Identity matrix I.
C
      INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2,
+       IEIDX1, IEIDX2, OEIDX1, OEIDX2, NUMEX, INDIM, OUTDIM, WTDIM,
+       LAYER(-1:LAIDX1, LAIDX2), UUIDX1, UUIDX2, UUIDX3
      DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2), NEURON(0:NUIDX1, NUIDX2),
+       INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2),
+       TOLERA, UUVV(0:UIDX1, 0:UIDX2, UUIDX3),
+       HESIAN(0:WTIDX1, 0:WTIDX1, 3)
C
      INTEGER P, EPOCH, NRANK, DIM, LDIM
      DOUBLE PRECISION RMS, RMSBAK, RR, LAMBDA, FACTOR, PSMAL, NORM,
+       INILAM, INIFAC, CUTOFF, SMALL
      COMMON /LEVVAR/INILAM, INIFAC, CUTOFF, SMALL
C
      PRINT *, 'Newton(I) is working'

```

```

C
  EPOCH=0
  LAMBDA=INILAM
  FACTOR=INIFAC
  PRINT *, 'FACTOR= ', FACTOR
C --- get initial RMS
  RMSBAK=0.0
  DO 2606 P=1, NUMEX
C
  setup inputs
  DO 2607 I=1, INDIM
    NEURON(I,1)=INEX(I,P)
2607  CONTINUE
  NEURON(0,1)=-1.0
C
  compute outputs of all neurons in the network
  CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+           NEURON, NUIDX1, NUIDX2)
C
  accumulate error over all patterns
  RMSBAK=RMSBAK+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+           OUTEX(1,P), OUTDIM)
2606  CONTINUE
  RMSBAK=DSQRT(RMSBAK/NUMEX)
  WRITE (*,2662) RMSBAK, EPOCH, LAMBDA, FACTOR
C
2601  DO 2605 J=0, WTDIM-1
    DO 2602 I=J, WTDIM-1
      HESIAN(I,J,3)=0.0
      HESIAN(J,I,3)=0.0
2602  CONTINUE
    WEIGHT(J,3)=0.0
2605  CONTINUE
C
C --- Batch model: accumulate error information over all patterns
  DO 2640 P=1, NUMEX
C
  setup inputs
  DO 2610 I=1, INDIM
    NEURON(I,1)=INEX(I,P)
2610  CONTINUE
  NEURON(0,1)=-1.0
C
C
  compute outputs of all neurons in the network
  CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+           NEURON, NUIDX1, NUIDX2)
C
C
  compute the gradient of E(w) for one pattern
  CALL COMGRA(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+           NEURON, NUIDX1, NUIDX2, OUTEX(1,P), OUTDIM)
C
  accumulate the NEGATIVE gradient over all patterns
  DO 2620 I=0, WTDIM-1
    WEIGHT(I,3)=WEIGHT(I,3) - WEIGHT(I,2)
2620  CONTINUE
C
C
  compute the useful partials
  CALL DUUVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+           NEURON, NUIDX1, NUIDX2, UUVV, UUIDX1, UUIDX2, UUIDX3)
  CALL DEVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+           NEURON, NUIDX1, NUIDX2,
+           UUVV, UUIDX1, UUIDX2, UUIDX3)
C
  compute the Hessian matrix
  CALL COMHAS(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+           UUVV, UUIDX1, UUIDX2, UUIDX3, HESIAN, WTIDX1)
C

```

```

C      sum up the Hessians over all patterns
      DO 2635 J=0, WTDIM-1
        DO 2630 I=J, WTDIM-1
          HESIAN(I,J,3)=HESIAN(I,J,3) + HESIAN(I,J,1)
          HESIAN(J,I,3)=HESIAN(I,J,3)
2630      CONTINUE
2635      CONTINUE
2640      CONTINUE
C
C --- formulate the linear system equations
      IF (LAMBDA .LT. CUTOFF) THEN
        LAMBDA=LAMBDA/(FACTOR*2.0)
      ELSE
        LAMBDA=LAMBDA/FACTOR
      ENDIF
C      add in Levenberg-Marquardt parameters
2645      DO 2655 J=0, WTDIM-1
        DO 2650 I=J, WTDIM-1
          HESIAN(I,J,2)=HESIAN(I,J,3)
          HESIAN(J,I,2)=HESIAN(I,J,3)
2650      CONTINUE
          WEIGHT(J,2)=WEIGHT(J,3)
          HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA
2655      CONTINUE
C
C --- setup parameters for calling LSOLVE
2661      DIM = WTDIM
          LDIM=WTIDX1+1
          CALL LSOLV(HESIAN(0,0,2),WEIGHT(0,2),DIM,LDIM,NRANK,PSMAL)
C
C      checking singularity
      IF (NRANK .LT. WTDIM) THEN
        PRINT*, 'singular system'
        LAMBDA=LAMBDA*FACTOR
        DO 2663 J=0, WTDIM-1
          DO 2664 I=J, WTDIM-1
            HESIAN(I,J,2)=HESIAN(I,J,3)
            HESIAN(J,I,2)=HESIAN(I,J,3)
2664      CONTINUE
            HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA
2663      CONTINUE
          GOTO 2661
        ENDIF
C
      NORM=0.0
C      temporary updating weights
      DO 2660 I=0, WTDIM-1
        NORM=NORM+WEIGHT(I,2)**2
        WEIGHT(I,2)=WEIGHT(I,1) + WEIGHT(I,2)
2660      CONTINUE
      NORM=DSQRT(NORM)
C
C      check learning result
      RMS=0.0
      DO 2670 P=1, NUMEX
C      setup inputs
        DO 2675 I=1, INDIM
          NEURON(I,1)=INEX(I,P)
2675      CONTINUE
          NEURON(0,1)=-1.0
C      compute outputs of all neurons in the network

```

```

        CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT(0,2), WTIDX1,
+           NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
        RMS=RMS+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+           OUTEX(1,P), OUTDIM)
2670 CONTINUE
C
        RMS=DSQRT(RMS/NUMEX)
        WRITE (*,2662) RMS, EPOCH, LAMBDA, NORM
2662 FORMAT('RMS=', F23.19, I10, ' LAMBDA=', 2G15.7)
        IF (RMS .GT. RMSBAK) THEN
            IF (LAMBDA .LT. CUTOFF) THEN
                LAMBDA=CUTOFF
            ELSE
                LAMBDA=LAMBDA*FACTOR
            ENDIF
            GOTO 2645
        ENDIF
C
        EPOCH=EPOCH+1
C      update weights
        DO 2680 I=0, WTDIM-1
            WEIGHT(I,1)=WEIGHT(I,2)
2680 CONTINUE
C      check if the convergence criterion is satisfied.
        IF (RMSBAK-RMS .GE. TOLERA) THEN
            RMSBAK=RMS
            GOTO 2601
        ENDIF
C      otherwise, training done
C
C*****
        RETURN
        END
C*****
C*****
SUBROUTINE GNLM(D, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+           NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+           OUTEX, OEIDX1, OEIDX2, NUMEX,
+           INDIM, OUTDIM, TOLERA,
+           UUVV, UUIDX1, UUIDX2, UUIDX3,
+           HESIAN, WTDIM)
C*****
C This routine implements the Gauss-Newton/Levenberg-Marquardt learning
C algorithm(Algorithm 3.5.1).
C
        INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2,
+           IEIDX1, IEIDX2, OEIDX1, OEIDX2, NUMEX, INDIM, OUTDIM, WTDIM,
+           LAYER(-1:LAIDX1, LAIDX2), UUIDX1, UUIDX2, UUIDX3
        DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2), NEURON(0:NUIDX1, NUIDX2),
+           INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2),
+           TOLERA, UUVV(0:UIDX1, 0:UIDX2, UUIDX3),
+           HESIAN(0:WTIDX1, 0:WTIDX1, 3)
C
        INTEGER P, EPOCH, NRANK, DIM, LDIM
        DOUBLE PRECISION RMS, RMSBAK, RR, LAMBDA, FACTOR, PSMAL, NORM,
+           INILAM, INIFAC, CUTOFF, SMALL
C
        COMMON /LEVLMAR/INILAM, INIFAC, CUTOFF, SMALL

        PRINT *, 'Gauss-Newton/Levenberg-Marquardt(D) is working'
        PRINT *, 'tolerance= ', TOLERA

```

```

C
  EPOCH=0
  LAMBDA=INILAM
  FACTOR=INIFAC
  PRINT *, 'FACTOR= ', FACTOR
C --- get initial RMS
  RMSBAK=0.0
  DO 2706 P=1, NUMEX
C    setup inputs
    DO 2707 I=1, INDIM
      NEURON(I,1)=INEX(I,P)
2707    CONTINUE
    NEURON(0,1)=-1.0
C    compute outputs of all neurons in the network
    CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+             NEURON, NUIDX1, NUIDX2)
C    accumulate error over all patterns
    RMSBAK=RMSBAK+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+             OUTEX(1,P), OUTDIM)
2706  CONTINUE
c    PRINT*, 'accumulated RMSBAK= ', RMSBAK
    RMSBAK=DSQRT(RMSBAK/NUMEX)
    WRITE (*,2762) RMSBAK, EPOCH, LAMBDA, FACTOR
C
2701  DO 2705 J=0, WTDIM-1
    DO 2702 I=J, WTDIM-1
      HESIAN(I,J,3)=0.0
      HESIAN(J,I,3)=0.0
2702    CONTINUE
    WEIGHT(J,3)=0.0
2705  CONTINUE
C
C --- Batch model: accumulate error information over all patterns
  DO 2740 P=1, NUMEX
C    setup inputs
    DO 2710 I=1, INDIM
      NEURON(I,1)=INEX(I,P)
2710    CONTINUE
    NEURON(0,1)=-1.0
C
C    compute outputs of all neurons in the network
    CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+             NEURON, NUIDX1, NUIDX2)
C
C    compute the gradient of E(w) for one pattern
    CALL COMGRA(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+             NEURON, NUIDX1, NUIDX2, OUTEX(1,P), OUTDIM)
C    accumulate the NEGATIVE gradient over all patterns
    DO 2720 I=0, WTDIM-1
      WEIGHT(I,3)=WEIGHT(I,3) - WEIGHT(I,2)
2720    CONTINUE
C
C    compute the useful partials
    CALL DUUVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+             NEURON, NUIDX1, NUIDX2, UUVV, UUIDX1, UUIDX2, UUIDX3)
c    compute the Jacobian matrix and estimate the Hessian
    CALL COMJAC(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+             NEURON, NUIDX1, NUIDX2, HESIAN, WTDIM,
+             UUVV, UUIDX1, UUIDX2, UUIDX3, OUTDIM)
C
C    sum up the Hessians over all patterns

```

```

        DO 2735 J=0, WTDIM-1
            DO 2730 I=J, WTDIM-1
                HESIAN(I,J,3)=HESIAN(I,J,3) + HESIAN(I,J,1)
                HESIAN(J,I,3)=HESIAN(I,J,3)
2730         CONTINUE
2735         CONTINUE
2740         CONTINUE
C
C --- formulate the linear system equations
IF ( LAMBDA .LT. CUTOFF ) THEN
    LAMBDA=LAMBDA/(FACTOR*2.0)
ELSE
    LAMBDA=LAMBDA/FACTOR
ENDIF
C
C add in Levenberg-Marquardt parameters
2745 DO 2755 J=0, WTDIM-1
        DO 2750 I=J, WTDIM-1
            HESIAN(I,J,2)=HESIAN(I,J,3)
            HESIAN(J,I,2)=HESIAN(I,J,3)
2750         CONTINUE
            WEIGHT(J,2)=WEIGHT(J,3)
            IF (HESIAN(J,J,3) .LT. SMALL) THEN
                HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA
            ELSE
                HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA*DABS(HESIAN(J,J,3))
            ENDIF
            HESIAN(J,J,2)=HESIAN(J,J,2) + LAMBDA
2755         CONTINUE
C --- setup parameters for calling LSOLVE
2757         FORMAT(20F20.10)
2761         DIM = WTDIM
            LDIM=WTIDX1+1
            CALL LSOLV(HESIAN(0,0,2),WEIGHT(0,2),DIM,LDIM,NRANK,PSMAL)
C checking singularity
IF (NRANK .LT. WTDIM) THEN
    PRINT*, 'singular system'
    LABMDA=FACTOR*LAMBDA
    DO 2763 J=0, WTDIM-1
        DO 2764 I=J, WTDIM-1
            HESIAN(I,J,2)=HESIAN(I,J,3)
            HESIAN(J,I,2)=HESIAN(I,J,3)
2764         CONTINUE
            HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA
2763         CONTINUE
            GOTO 2761
        ENDIF
C
C temporary updating weights
NORM=0.0
DO 2660, I=0, WTDIM-1
    NORM=NORM+WEIGHT(I,2)**2
    WEIGHT(I,2)=WEIGHT(I,1) + WEIGHT(I,2)
2660 CONTINUE
NORM=DSQRT(NORM)
C
C check learning result
RMS=0.0
DO 2770 P=1, NUMEX
C setup inputs
DO 2775 I=1, INDIM

```

```

                NEURON(I,1)=INEX(I,P)
2775    CONTINUE
        NEURON(0,1)=-1.0
C      compute outputs of all neurons in the network
        CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT(0,2), WTIDX1,
+           NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
        RMS=RMS+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+           OUTEX(1,P), OUTDIM)
c      PRINT*, 'accumulating RMS=', RMS
2770    CONTINUE
C
C      PRINT *, 'accumulated RMS= ', RMS
        RMS=DSQRT(RMS/NUMEX)
        WRITE (*,2762) RMS, EPOCH, LAMBDA, NORM
2762    FORMAT('RMS=', F23.19, I10, ' LAMBDA=', 2G15.7)
        IF (RMS .GT. RMSBAK ) THEN
            IF (LAMBDA .LT. CUTOFF) THEN
                LAMBDA=CUTOFF
            ELSE
                LAMBDA=LAMBDA*FACTOR
            ENDIF
            GOTO 2745
        ENDIF
C
        EPOCH=EPOCH+1
C      update weights
        DO 2780 I=0, WTDIM-1
            WEIGHT(I,1)=WEIGHT(I,2)
2780    CONTINUE
C      check if the convergence criterion is satisfied.
        IF (RMSBAK-RMS .GE. TOLERA) THEN
            RMSBAK=RMS
            GOTO 2701
        ENDIF
C      otherwise, training done
C
C*****
        RETURN
        END
C*****
C*****
SUBROUTINE NLMD(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+           NEURON, NUIDX1, NUIDX2, INEX, IEIDX1, IEIDX2,
+           OUTEX, OEIDX1, OEIDX2, NUMEX,
+           INDIM, OUTDIM, TOLERA,
+           UVV, UUIDX1, UUIDX2, UUIDX3,
+           HESIAN, WTDIM)
C*****
C This routine implements the Damped Newton learning algorithm
C (Algorithm 4.2.1).
C
        INTEGER LAIDX1, LAIDX2, WTIDX1, WTIDX2, NUIDX1, NUIDX2,
+           IEIDX1, IEIDX2, OEIDX1, OEIDX2, NUMEX, INDIM, OUTDIM, WTDIM,
+           LAYER(-1:LAIDX1, LAIDX2), UUIDX1, UUIDX2, UUIDX3
        DOUBLE PRECISION WEIGHT(0:WTIDX1, WTIDX2), NEURON(0:NUIDX1, NUIDX2),
+           INEX(IEIDX1, IEIDX2), OUTEX(OEIDX1, OEIDX2),
+           TOLERA, UVV(0:UUIX1, 0:UUIX2, UUIX3),
+           HESIAN(0:WTIDX1, 0:WTIDX1, 3)
C
        INTEGER P, EPOCH, NRANK, DIM, LDIM

```

```

DOUBLE PRECISION RMS, RMSBAK, RR, LAMBDA, FACTOR, PSMAI, NORM,
+      INILAM, INIFAC, CUTOFF, SMALL
COMMON /LEVMAR/INILAM, INIFAC, CUTOFF, SMALL

C
PRINT *, 'Newton(D) is working'
C
EPOCH=0
LAMBDA=INILAM
FACTOR=INIFAC
PRINT *, 'FACTOR= ', FACTOR
C --- get initial RMS
RMSBAK=0.0
DO 2806 P=1, NUMEX
C      setup inputs
DO 2807 I=1, INDIM
      NEURON(I,1)=INEX(I,P)
2807 CONTINUE
      NEURON(0,1)=-1.0
C      compute outputs of all neurons in the network
CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+          NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
RMSBAK=RMSBAK+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+          OUTEX(1,P), OUTDIM)
2806 CONTINUE
RMSBAK=DSQRT(RMSBAK/NUMEX)
WRITE (*,2862) RMSBAK, EPOCH, LAMBDA, FACTOR
C
2801 DO 2805 J=0, WTDIM-1
      DO 2802 I=J, WTDIM-1
          HESIAN(I,J,3)=0.0
          HESIAN(J,I,3)=0.0
2802 CONTINUE
      WEIGHT(J,3)=0.0
2805 CONTINUE
C
C --- Batch model: accumulate error information over all patterns
DO 2840 P=1, NUMEX
C      setup inputs
DO 2810 I=1, INDIM
      NEURON(I,1)=INEX(I,P)
2810 CONTINUE
      NEURON(0,1)=-1.0
C
C      compute outputs of all neurons in the network
CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1,
+          NEURON, NUIDX1, NUIDX2)
C
C      compute the gradient of E(w) for one pattern
CALL COMGRA(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+          NEURON, NUIDX1, NUIDX2, OUTEX(1,P), OUTDIM)
C      accumulate the NEGATIVE gradient over all patterns
DO 2820 I=0, WTDIM-1
      WEIGHT(I,3)=WEIGHT(I,3) - WEIGHT(I,2)
2820 CONTINUE
C
C      compute the useful partials
CALL DUUVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+          NEURON, NUIDX1, NUIDX2, UUVV, UUIDX1, UUIDX2, UUIDX3)
CALL DEVV(LAYER, LAIDX1, LAIDX2, WEIGHT, WTIDX1, WTIDX2,
+          NEURON, NUIDX1, NUIDX2,

```



```

+          UUVV, UUIDX1, UUIDX2, UUIDX3)
C      compute the Hessian matrix
      CALL COMHAS(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+          UUVV, UUIDX1, UUIDX2, UUIDX3, HESIAN, WTIDX1)
C
C      sum up the Hessians over all patterns
      DO 2835 J=0, WTDIM-1
          DO 2830 I=J, WTDIM-1
              HESIAN(I,J,3)=HESIAN(I,J,3) + HESIAN(I,J,1)
              HESIAN(J,I,3)=HESIAN(I,J,3)
2830          CONTINUE
2835      CONTINUE
2840  CONTINUE
C
C --- formulate the linear system equations
      IF (LAMBDA .LT. CUTOFF) THEN
          LAMBDA=LAMBDA/(FACTOR*2.0)
      ELSE
          LAMBDA=LAMBDA/FACTOR
      ENDIF
C      add in Levenberg-Marquardt parameters
2845  DO 2855 J=0, WTDIM-1
          DO 2850 I=J, WTDIM-1
              HESIAN(I,J,2)=HESIAN(I,J,3)
              HESIAN(J,I,2)=HESIAN(I,J,3)
2850      CONTINUE
          WEIGHT(J,2)=WEIGHT(J,3)
          IF (HESIAN(J,J,3) .LT. SMALL) THEN
              HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA
          ELSE
              HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA*DABS(HESIAN(J,J,3))
          ENDIF
2855  CONTINUE
C
C --- setup parameters for calling LSOLVE
2861  DIM = WTDIM
      LDIM=WTIDX1+1
      CALL LSOLV(HESIAN(0,0,2),WEIGHT(0,2),DIM,LDIM,NRANK,PSMAL)
C
C      checking singularity
      IF (NRANK .LT. WTDIM) THEN
          PRINT*, 'singular system'
          LAMBDA=LAMBDA*FACTOR
          DO 2863 J=0, WTDIM-1
              DO 2864 I=J, WTDIM-1
                  HESIAN(I,J,2)=HESIAN(I,J,3)
                  HESIAN(J,I,2)=HESIAN(I,J,3)
2864          CONTINUE
              HESIAN(J,J,2)=HESIAN(J,J,3)+LAMBDA
2863      CONTINUE
          GOTO 2861
      ENDIF
C
      NORM=0.0
C      temporary updating weights
      DO 2860 I=0, WTDIM-1
          NORM=NORM+WEIGHT(I,2)**2
          WEIGHT(I,2)=WEIGHT(I,1) + WEIGHT(I,2)
2860  CONTINUE
      NORM=DSQRT(NORM)
C

```

```

C      check learning result
      RMS=0.0
      DO 2870 P=1, NUMEX
C      setup inputs
      DO 2875 I=1, INDIM
          NEURON(I,1)=INEX(I,P)
2875  CONTINUE
      NEURON(0,1)=-1.0
C      compute outputs of all neurons in the network
      CALL NETOUT(LAYER, LAIDX1, LAIDX2, WEIGHT(0,2), WTIDX1,
+          NEURON, NUIDX1, NUIDX2)
C      accumulate error over all patterns
      RMS=RMS+RR(LAYER, LAIDX1, LAIDX2, NEURON, NUIDX1, NUIDX2,
+          OUTEX(1,P), OUTDIM)
2870  CONTINUE
C
      RMS=DSQRT(RMS/NUMEX)
      WRITE (*,2862) RMS, EPOCH, LAMBDA, NORM
2862  FORMAT('RMS=', F23.19, I10, ' LAMBDA=',2G15.7)
      IF (RMS .GT. RMSBAK) THEN
          IF (LAMBDA .LT. CUTOFF) THEN
              LAMBDA=CUTOFF
          ELSE
              LAMBDA=LAMBDA*FACTOR
          ENDIF
          GOTO 2845
      ENDIF
C
      EPOCH=EPOCH+1
C      update weights
      DO 2880 I=0, WTDIM-1
          WEIGHT(I,1)=WEIGHT(I,2)
2880  CONTINUE
C      check if the convergence criterion is satisfied.
      IF (RMSBAK-RMS .GE. TOLERA) THEN
          RMSBAK=RMS
          GOTO 2801
      ENDIF
C      otherwise, training done
C
C*****
      RETURN
      END
C*****

```

VITA

LIYA WANG

Candidate for the Degree of
Master of Science

Thesis: THE DAMPED NEWTON METHOD
--AN ANN LEARNING ALGORITHM

Major Field: Computer Science

Biographical:

Personal Date: Born in Beijing, P.R. of China, on October 31, 1963.

Education: Received Bachelor of Science degree in Automation and Computer Science from China University of Mining and Technology in July, 1985; obtained Master of Science degree in Mathematics from Northern Arizona University in May, 1993. completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in December 1995.

Experience: Employed as a programmer by China Research Institute of Printing Science and Technology from September 1985 to June 1990.