

A PENALTY METHOD TO REDUCE OVERFITTING
IN ARTIFICIAL NEURAL NETWORKS

By

PING JIANG

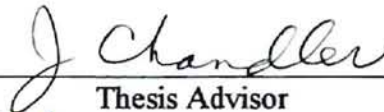
Bachelor of Science
Tongji University
Shanghai, PR China
1984

Master of Science
Oklahoma State University
Slitter, Oklahoma
1994

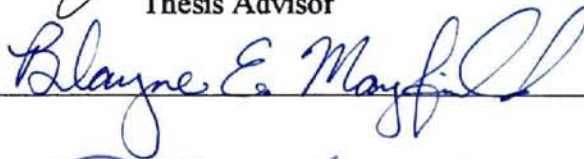
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1996

A PENALTY METHOD TO REDUCE OVERFITTING
IN ARTIFICIAL NEURAL NETWORKS

Thesis Approved:



Thesis Advisor





Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. John P. Chandler for his guidance, encouragement and invaluable instructions. He made a great effort to improve this thesis in both contents and English. Likewise, sincere appreciation is extended to the advisory committee members, Dr. K.M. George and Dr. B. E. Mayfield, for their assistance.

Very special thanks and love go to my parents, Su Zhen Zhu and Yun Jiang. I would also like to express my gratitude to my sister Wan Qing Jiang and her family for their continuing support and understanding during my study overseas. Without their support, this thesis would not be possible.

Finally I would like to express thanks to all faculty and staff in the Computer Science Department for their support during my study in Oklahoma State University.

TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION	1
1.1 Artificial Neural Network History	1
1.2 Artificial Neural Network Models and Applications	2
1.3 Learning in Feedforward Artificial Neural Networks	5
1.4 Overfitting in Artificial Neural Networks	8
1.5 The Objective of This Study	9
2. PERFORMANCE OPTIMIZATION	11
2.1 Basic Concepts	11
2.2 Steepest Descent Method	19
2.3 Newton's Method	22
2.4 The Conjugate Gradient Method	24
3. ARTIFICIAL NEURAL NETWORK LEARNING ALGORITHMS	28
3.1 Architectures of Feedforward Artificial Neural Networks	28
3.2 Dynamic Behavior in Artificial Neural Networks	34
3.3 Overfitting and Generalization in Artificial Neural Networks	36
3.4 Stopped Training Method to Reduce Overfitting	38
3.5 Penalty Method to Reduce Overfitting	39

3.6 Computation in Feedforward Artificial Neural Networks	41
4. IMPLEMENTATION AND DISCUSSION OF RESULTS	49
4.1 Language Implementation and Neural Network Architecture Design	49
4.2 Discussion of Test Results	50
5. Conclusion and Future Work	75
BIBLIOGRAPHY	77
APPENDIX -- PROGRAM LISTING	82

LIST OF TABLES

Table	Page
4.1 Performance of Training and Generalization RMS with 7 hidden nodes and $\lambda = 0$	54
4.2 Performance of Training and Generalization RMS with 7 hidden nodes and $\lambda = 0.00001$	54
4.3 Performance of Training and Generalization RMS with 7 hidden nodes and $\lambda = 0.0001$	55
4.4 Performance of Training and Generalization RMS with 7 hidden nodes and $\lambda = 0.001$	55
4.5 Performance of Training and Generalization RMS with 7 hidden nodes and $\lambda = 0.01$	55
4.6 Performance of Training and Generalization RMS with 8 hidden nodes and $\lambda = 0$	56
4.7 Performance of Training and Generalization RMS with 8 hidden nodes and $\lambda = 0.00001$	56
4.8 Performance of Training and Generalization RMS with 8 hidden nodes and $\lambda = 0.0001$	57
4.9 Performance of Training and Generalization RMS with 8 hidden nodes and $\lambda = 0.001$	57
4.10 Performance of Training and Generalization RMS with 8 hidden nodes and $\lambda = 0.01$	57
4.11 Performance of Training and Generalization RMS with 9 hidden nodes and $\lambda = 0$	58
4.12 Performance of Training and Generalization RMS with 9 hidden nodes and $\lambda = 0.00001$	58
4.13 Performance of Training and Generalization RMS with 9 hidden nodes and $\lambda = 0.0001$	59

4.14 Performance of Training and Generalization RMS with 9 hidden nodes and $\lambda = 0.001$	59
4.15 Performance of Training and Generalization RMS with 9 hidden nodes and $\lambda = 0.01$	60
4.16 Performance of Training and Generalization RMS with 10 hidden nodes and $\lambda = 0$	60
4.17 Performance of Training and Generalization RMS with 10 hidden nodes and $\lambda = 0.00001$	61
4.18 Performance of Training and Generalization RMS with 10 hidden nodes and $\lambda = 0.0001$	61
4.19 Performance of Training and Generalization RMS with 10 hidden nodes and $\lambda = 0.001$	62
4.20 Performance of Training and Generalization RMS with 10 hidden nodes and $\lambda = 0.01$	62
4.21 Performance of Training and Generalization RMS with 17 hidden nodes and $\lambda = 0$	62
4.22 Performance of Training and Generalization RMS with 17 hidden nodes and $\lambda = 0.00001$	63
4.23 Performance of Training and Generalization RMS with 17 hidden nodes and $\lambda = 0.0001$	63
4.24 Performance of Training and Generalization RMS with 17 hidden nodes and $\lambda = 0.001$	63
4.25 Performance of Training and Generalization RMS with 17 hidden nodes and $\lambda = 0.01$	64
4.26 Performance of Training and Generalization RMS with 18 hidden nodes and $\lambda = 0$	64
4.27 Performance of Training and Generalization RMS with 18 hidden nodes and $\lambda = 0.00001$	64
4.28 Performance of Training and Generalization RMS with 18 hidden nodes and $\lambda = 0.0001$	65

4.29 Performance of Training and Generalization RMS with 18 hidden nodes and $\lambda = 0.001$	65
4.30 Performance of Training and Generalization RMS with 18 hidden nodes and $\lambda = 0.01$	65
4.31 Performance of Training and Generalization RMS with 19 hidden nodes and $\lambda = 0$	66
4.32 Performance of Training and Generalization RMS with 19 hidden nodes and $\lambda = 0.00001$	66
4.33 Performance of Training and Generalization RMS with 19 hidden nodes and $\lambda = 0.0001$	66
4.34 Performance of Training and Generalization RMS with 19 hidden nodes and $\lambda = 0.001$	67
4.35 Performance of Training and Generalization RMS with 19 hidden nodes and $\lambda = 0.01$	67
4.36 Performance of Training and Generalization RMS with 20 hidden nodes and $\lambda = 0$	67
4.37 Performance of Training and Generalization RMS with 20 hidden nodes and $\lambda = 0.0001$	68
4.38 Performance of Training and Generalization RMS with 20 hidden nodes and $\lambda = 0.001$	68
4.39 Performance of Training and Generalization RMS with 20 hidden nodes and $\lambda = 0.01$	69

LIST OF FIGURES

Figure	Page
1.2.1 A Generic Processing Element	3
3.1.1 A Three Layer Feedforward Network	29
3.1.2 The Sigmoid Function	33
3.1.3 The Hyperbolic Function	33
3.3.1 The Relationship Between Training Error and Testing Error	36
3.5.1 The First Order Derivative of Penalty Term	41
4.1 The Relationship Between Generalization RMS and λ (2/7/1)	70
4.2 The Relationship Between Generalization RMS and λ (2/8/1)	70
4.3 The Relationship Between Generalization RMS and λ (2/9/1)	71
4.4 The Relationship Between Generalization RMS and λ (2/10/1)	71
4.5 The Relationship Between Generalization RMS and λ (2/17/1)	72
4.6 The Relationship Between Generalization RMS and λ (2/18/1)	72
4.7 The Relationship Between Generalization RMS and λ (2/19/1)	73
4.8 The Relationship Between Generalization RMS and λ (2/20/1)	73
4.9 The Relationship Between Generalization RMS and Number of Weights	74

1. INTRODUCTION

1.1 Artificial Neural Network History

Neurocomputing is an interdisciplinary concerned with information processing systems, i.e., neural networks that can be trained to develop operational capabilities to respond to an information environment. The human brain is composed of about 10^{11} neurons (nerve cells) of different types [1]. The neural network was originally aimed towards modeling networks of real neuron in the brain. The history of neural networks can be traced back to 1943 when Warren McCulloch and Walter Pitts [2][3] proposed a simple model of a neuron as a binary threshold unit to compute arithmetic and logical functions. In 1949 Donald Hebb [4] published a book called "The Organization of Behavior" which proposed a specific learning law for the synapses of neurons. He used this learning law to explain qualitatively some experimental results from psychology. Hebb's research inspired many researchers to pursue the same theme, which eventually laid down the foundation for the advent of neurocomputing. The first successful neurocomputer called the Mark I Perceptron, was built by Frank Rosenblatt, Charles Wightman and others during 1957 and 1958 [6]. Rosenblatt is considered to be the founder of neurocomputing. Bernard Widrow, working with his graduate students, developed different types of neural network processing elements called the ADALINE and MADALINE, and applied them successfully in a type of electronically adjustable resistor called the memistor. Despite some setbacks in the late 1960s and 1970s, the artificial neural network researches regained their momentum thanks to physicist John Hopfield and other dedicated researchers. In 1986, David Rumelhart and James McClelland edited a

book called Parallel Distributed Processing (PDP) [20], Volume I and Volume II. The field exploded since then. Today we are witnessing substantial growth in neural network research and development.

1.2 Artificial Neural Network Models and Applications

A neural network is a parallel distributed information processing structure in the form of a directed graph, with the following sub-definitions [2]

- The nodes of the graph are called processing elements or artificial neurons.
- The links of the graph are called connections.
- Each processing element can receive any number of incoming connections (also called “put” connections).
- Each processing element can have any number of outgoing connections.
- Processing elements can have local memory.
- Each processing element processes a transfer function which can use (and alter) local memory, use input signals, and produce output signals.

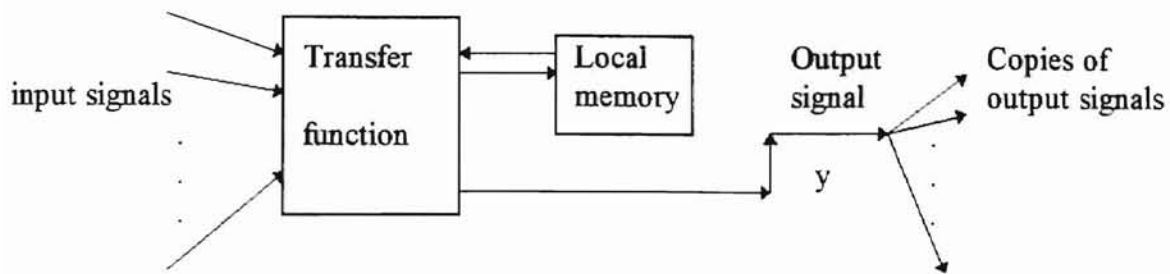


Figure 1.2.1 A Generic Processing Element

A generic processing element is shown in Figure 1.2.1. A typical neural network consists of many simple neuron-like processing elements, also called units or neurons. These processing elements are linked together to form a network. There are many different ways to connect the processing elements. Therefore there are many different neural network models. Basically we can divide neural network models into three categories [15]:

- Feedforward (multilayer) networks
- Feedback (recurrent) networks
- Cellular networks

In feedforward neural networks, processing elements are arranged in a feedforward manner. One example of feedforward networks is the fully connected feedforward network model. On the other hand, in feedback neural networks, the models are no longer trivial since they consist of processing elements with dynamic building blocks (e.g. integrator or unit delays) and they operate in feedback mode. The Hopfield network and Boltzmann machine are examples of feedback networks. Cellular neural networks, similar to cellular automata, consist of regularly spaced special artificial neurons

called cells, which communicate directly with other neurons only in their nearest neighborhood. The Kohonen map is one example of cellular networks.

Neural networks have been used in many fields. A list of some applications are as follows [16]:

Transportation: Aircraft control systems, automobile automatic guiding systems.

Economics: Credit card application processing, corporation financial analysis, currency price prediction, market forecasting.

Defense: Weapon steering, target tracking, object discrimination, signal/image identification and data compression.

Electronics: Code sequence prediction, process control, chip failure analysis, voice synthesis.

Manufacturing: Manufacturing process control, visual quality inspection systems, product quality prediction.

Medical: Optimization of transplant times, hospital quality improvement.

Robotics: Trajectory control, manipulator controls, vision systems.

Speech: Speech recognition, speech compression.

Telecommunications: Image and data compression, automated information services.

1.3 Learning in Feedforward Artificial Neural Networks

One of the interesting properties of a feedforward neural network is its capability of learning, i.e., a feedforward neural network can self-adjust its behavior by using information from the environment. When we use a feedforward neural network to solve a problem, we first train the network using a set of input-output sample data. Based on this data set, the network, when properly trained, will not only try to learn the sample set correctly, but also to generalize from the training set examples to the entire problem domain. This capability of generalization makes artificial neural networks very useful tools to solve a set of problems which are not clearly defined.

A neural network consists of processing elements and connections. Each connection has a weight to represent the relative importance of the connection, except for the output connections. A processing element sums all its weighted inputs and passes the result to the transfer (activation) function to yield the output of the processing element. The nonlinear transfer function can be a step (threshold) function as used by McCulloch-Pitts[3] to solve classification problems. But generally the step function is replaced by a non-linear continuous function (e.g. a sigmoid function). Artificial neural networks are organized into layers. A neural network links the output of neurons of one layer to the neurons of the next layer. A fully connected feedforward network is a network such that the output of neurons in one layer are linked to all neurons of the next layer, except for the output layer of the network. The computation of the network is carried out on a layer-by-layer basis, starting from the input layer. The computation process continues until the output has been reached. Such a computation is called a forward pass.

Learning is an important requirement of neural networks. A neural network usually has to be trained to perform a desired task. The application of neural networks involves two major phases: learning phase and performance phase. During the learning phase, a neural network is given a set of input/output sample data. The network calculates the output based on the input data and the result is compared with the desired output. If the calculated output is not close to the desired output, the network will try to modify its weights until a better approximation is reached. Such learning is called supervised learning. This learning method is also called "learning with a teacher" because the learning is done on the basis of direct comparison of network output with known correct answers. Sometimes the learning goal is not defined at all in terms of specific correct examples. The only available information is in the correlation of the input data or signals. The network is expected to create categories from the correlation and to produce outputs corresponding to the input category. Such a learning is called unsupervised learning. Often the learning phase will involve hundreds of thousands of repetitions as the neural network goes through all of the training examples before the neural network enters the performance phase. During the performance phase, the neural network is able to compute outputs from non-example input data.

From an optimization point of view, learning in a neural network is equivalent to minimizing the sum of squares of the output errors, sometimes called "error function". A learning algorithm is applied to transform the calculated errors into weight adjustments until a local minimum in the error function is reached. Most learning algorithms that are

used for training feedforward neural networks are those that enforce the learning process by means of backpropagation [10].

Although the learning algorithms may be different, the learning procedures are basically the same. The following is a general outline of the learning procedure used in all backpropagation algorithms:

1. The network is given a random set of initial weights.
2. Training examples are given to the network.
3. For each input-output pairs, there are two phases: a forward pass and a backward pass.
4. The forward pass is to calculate the outputs on a layer-by-layer basis until the output layer has been reached.
5. During the backward pass, the calculated outputs from the output layer are compared with the desired output, and errors are computed for the output nodes. Then the network adjusts its weights in a backward fashion, starting from the output layer, to reduce the errors.
6. This process continues until convergence has been reached.

There are several mathematical models in optimization [12][14][15] which can be applied in the learning process of artificial neural networks. The least mean square (LMS) model is the most widely used in artificial neural network analysis. The error (performance) function is defined as the squared summation of the difference between the computed outputs and the desired outputs. The optimization goal is to minimize this error function. Most of the learning algorithms are gradient-based learning algorithm which can

be divided into three categories: the steepest descent method, the Newton method and the conjugate gradient methods[12]. These methods will be introduced in Chapter two.

1.4. Overfitting in Artificial Neural Networks

When a neural network is trained, the weights are modified in order to minimize the error in the training patterns. For continuous domains, or large discrete ones, it is impossible to provide samples of every possible input. For a large network system, if the system simply memorizes the training patterns, it may do quite well during the training process but it may give spurious and misleading outputs if the input is slightly different from the sample inputs. This is called overfitting.[23] An example is a high-order polynomial fitted through a small number of points. Overfitting happens when the network has as many or more degrees of freedom (the number of weights) than the number of training samples. In other words there are not enough examples to constrain the network. It is advisable to use the smallest system that will fit the data. If the system has only a limited number of degrees of freedom, it will use a limited number of data to adapt to the largest constraints and ignore the smaller (possible spurious) constraints. As a rule of thumb, for a network to be able to generalize, it should have fewer parameters(weights) than there are data points in the training set. Unfortunately, it usually isn't obvious what size is best so a common approach is to train successively smaller networks until the smallest one is found that will learn the data.[23][27] This approach has several disadvantages. First it is time consuming, since a large number of networks must be trained. Second, the smallest feasible networks may be sensitive to initial conditions and learning parameters, and be more likely to be trapped in local minima. Another approach is

to have the network itself remove non-useful connections during training by giving each connection a tendency to decay, so that connections disappear [23][24][26]. This prompts me to use penalty method to reduce overfitting in artificial neural networks [24][26]. The purpose of this thesis will be illustrated in the following section.

1.5. The Objective of This Study

This thesis focuses on the possibilities of reducing overfitting in artificial neural networks. The penalty method[24][26] will be implemented to reduce overfitting. The idea is to add a term to the performance function as follows:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^Q \left((\mathbf{f}^{[K]}(\mathbf{p}_i, \mathbf{w}) - \mathbf{t}_i)^T (\mathbf{f}^{[K]}(\mathbf{p}_i, \mathbf{w}) - \mathbf{t}_i) + \lambda \sum_j \frac{w_j^2 / w_0^2}{1 + w_j^2 / w_0^2} \right) \quad (1.5.1)$$

The first term measures the performance of the network. It is the sum of squared errors over the set of training data. The second term measures the size of the network. Its sum extends over all connections C . λ represents the relative importance of the complexity term with respect to the performance term.

The learning rule is to change the weights according to the gradient of the entire function, continuously doing justice to the trade-off between error and complexity. The extreme cases of very large and very small weights are easily interpreted. For $|w_i| \gg w_0$, the second term is close to λ . This justifies the interpretation of the complexity term as a counter of significant-sized weights. On the other hand, if $|w_i| \ll w_0$ the second term is close to zero. "Large" and "Small" are defined with respect to the scale w_0 , a constant parameter that has to be decided in the procedure. Q is the number of input output pairs.

The conjugate gradient method[12][14][15] will be used to minimize the performance function (1.5.1). To fully understand the subject, we need some basic knowledge of nonlinear optimization, which will be discussed in Chapter two.

2. PERFORMANCE OPTIMIZATION

From an optimization point of view, training a network is equivalent to minimizing a global error function, which is a multivariate function that depends on the weights in the network. In this chapter, we introduce some fundamental concepts, various classical optimization methods and the mathematical principles behind these methods. The principles were discovered by scientists and mathematicians such as Kepler, Newton, Gauss, Cauchy and Leibniz. With the advent of digital computers, these principles have been successfully applied to develop algorithms in the field of optimization. Although these algorithms are different, they all use iterative methods. We will first introduce some basic concepts that will be used later in analyses of various optimization methods.

2.1 Basic Concepts [11]

Unless otherwise stated, the domains of all variables are real numbers.

An $m \times n$ matrix represented by

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdot & \cdot & \cdot & a_{1n} \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ a_{m1} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix} \quad (2.1.1)$$

can be expressed as $\mathbf{A} = [a_{ij}]_{m \times n}$. It has m rows and n columns. If m is equal to n , the matrix is called a square matrix. A $1 \times n$ matrix is called a row vector and an $m \times 1$ matrix is called a column vector. In this thesis, all matrices are represented by uppercase bold face letters and row vectors are represented by lowercase bold face letters.

The multiplication of two matrices $\mathbf{A} = [a_{ij}]_{m \times n}$ and $\mathbf{B} = [b_{jk}]_{n \times l}$ is defined as

$\mathbf{C} = [c_{ik}]_{m \times l}$ such that

$$c_{ik} = \sum_{j=1}^n a_{ij} \times b_{jk} \quad (2.1.2)$$

The number of columns in matrix \mathbf{A} must be equal to the number of rows in matrix \mathbf{B} .

The transpose of a matrix $\mathbf{A} = [a_{ij}]_{m \times n}$ represented by $\mathbf{A}^T = [a_{ji}]_{n \times m}$ is a matrix such that $a_{ij} = a_{ji}$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$. Obviously we have

$$(\mathbf{A}^T)^T = \mathbf{A} \quad (2.1.3)$$

and also

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T \quad (2.1.4)$$

A matrix $\mathbf{A} = [a_{ij}]_{n \times n}$ is called symmetric if it is a square matrix such that $a_{ij} = a_{ji}$

for all $1 \leq i \leq n$ and $1 \leq j \leq n$. It can be represented as

$$\mathbf{A} = \mathbf{A}^T \quad (2.1.5)$$

A diagonal matrix $\mathbf{A} = [a_{ij}]_{n \times n}$ is a square matrix such that $a_{ij} = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq n$ and $i \neq j$ and is represented by

$$\mathbf{D} = \text{diag}(a_{11}, a_{22}, \dots, a_{nn}) \quad (2.1.6)$$

Specifically if $a_{ii} = 1$ for all $1 \leq i \leq n$, then the diagonal matrix is called an identity matrix

I.

A square matrix \mathbf{B} is called the inverse of a square matrix of \mathbf{A} if

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I} \quad (2.1.7)$$

\mathbf{B} can be written as \mathbf{A}^{-1} . If the inverse of a matrix \mathbf{A} doesn't exist, \mathbf{A} is called a singular matrix. Otherwise it is called nonsingular.

A symmetric matrix \mathbf{A} is called positive (negative) definite if the quadratic form

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0 \quad (< 0) \text{ for all } \mathbf{x} \neq 0 \quad (2.1.8)$$

A matrix \mathbf{A} is called positive (negative) semidefinite if the equality is included in the above condition.

A simultaneous linear equation system, represented by

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ \cdots \cdots \cdots \cdots \cdots \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \quad (2.1.9)$$

can be written as $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} = [a_{ij}]_{m \times n}$, $\mathbf{x} = [x_i]_{n \times 1}$ and $\mathbf{b} = [b_i]_{m \times 1}$. The equation is solvable if \mathbf{A}^{-1} exists. In that case

$$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b} \quad (2.1.10)$$

The Gaussian elimination method can be used to solve the system (2.1.10).

The inner product of two vectors $\mathbf{x} = [x_i]_{n \times 1}$ and $\mathbf{y} = [y_i]_{n \times 1}$ is defined as

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \mathbf{y}^T \mathbf{x} = \sum_1^n x_i y_i \quad (2.1.11)$$

The k-norm of a vector $\mathbf{x} = [x_i]_{n \times 1}$ is defined as

$$\|\mathbf{x}\|_k = \left(\sum_1^n |x_i|^k \right)^{1/k} \quad (2.1.12)$$

Specifically when k is equal to 2, it is called the Euclidean norm. It represents the length of an n -dimensional space vector. For Euclidean normed vectors, there is a Schwarz inequality

$$|\langle \mathbf{x}, \mathbf{y} \rangle| = |\mathbf{x}^T \mathbf{y}| \leq \|\mathbf{x}\|_2^{1/2} \|\mathbf{y}\|_2^{1/2} \quad (2.1.13)$$

The equality holds if and only if $\mathbf{x} = \lambda \mathbf{y}$, λ is real number.

Two vectors \mathbf{x} and \mathbf{y} are said to be orthogonal if $\langle \mathbf{x}, \mathbf{y} \rangle = 0$

A transformation \mathbf{T} from \mathbf{x} to \mathbf{y} ($\mathbf{T}: \mathbf{x} \rightarrow \mathbf{y}$) consists of three things

- (1) A set of elements $x_i \in \mathbf{x}$ called the domain
- (2) A set of elements $y_i \in \mathbf{y}$ called the range.
- (3) A rule relating each $x_i \in \mathbf{x}$ to $y_i \in \mathbf{y}$.

A transformation is called linear if

- (1) For all $x_i, x_j \in \mathbf{x}$,

$$\mathbf{T}(x_i + x_j) = \mathbf{T}(x_i) + \mathbf{T}(x_j) \quad (2.1.14)$$

- (2) For all $x_i \in \mathbf{x}$, $a \in \mathbf{R}$,

$$\mathbf{T}(ax_i) = a\mathbf{T}(x_i) \quad (2.1.15)$$

Linear independence

Consider a set of vectors $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_n\}$, If there exist n scalars a_1, a_2, \dots, a_n , at least one of which is nonzero, such that

$$a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \dots + a_n \mathbf{x}_n = 0 \quad (2.1.16)$$

then the set of n vectors are called linearly independent. Let X be a linear vector space, and let $\{x_1, x_2, x_3, \dots, x_n\}$ be a subset of vectors in x . This subset spans X if and only if every vector $x \in X$, there exist n scalars a_1, a_2, \dots, a_n , such that

$$X = a_1x_1 + a_2x_2 + \dots + a_nx_n \quad (2.1.17)$$

A basis set for X is a set of linearly independent vectors which spans X . Although any vector space can have many basis sets, the number of elements in basis sets is the same [12]. Given n independent vectors x_1, x_2, \dots, x_n , we can obtain n orthogonal vectors y_1, y_2, \dots, y_n by the Gram-Schmidt method as follows

$$\begin{aligned} y_1 &= x_1 \\ y_k &= x_k - \sum_{i=1}^{k-1} \frac{(y_i, x_k)}{(y_i, y_i)} \cdot y_i \end{aligned} \quad 2 \leq k \leq n \quad (2.1.18)$$

Eigenvalues and Eigenvectors

Consider a linear transformation $A: X \rightarrow X$, Given a set of vectors $z \in X$ which are not equal to zero, and a set of scalars λ that satisfy

$$A(z) = \lambda z \quad (2.1.19)$$

z and λ are called eigenvectors and eigenvalues, respectively. The matrix representation of the eigencharacter equation is

$$Az = \lambda z \quad (2.1.20)$$

or

$$[A - \lambda I] z = 0 \quad (2.1.21)$$

Because $\mathbf{z} \neq \mathbf{0}$ we can obtain the eigenvalues and eigenvectors by solving the following equation

$$[(\mathbf{A} - \lambda \mathbf{I})] = \mathbf{0} \quad (2.1.22)$$

If we have n distinct eigenvalues for an n -dimensional matrix \mathbf{A} , we are guaranteed to find n independent eigenvectors. Therefore the eigenvectors make up a basis set for the vector space of the transformation. Furthermore, let $\mathbf{B} = [\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n]$, where $\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n$ are the eigenvectors of the matrix \mathbf{A} . We have the following diagonalization.

$$[\mathbf{B}^{-1} \mathbf{A} \mathbf{B}] = \begin{bmatrix} \lambda_1 & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & \lambda_2 & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & & & & \cdot \\ \cdot & \cdot & & & & \cdot \\ \cdot & \cdot & & & & \cdot \\ 0 & 0 & \cdot & \cdot & \cdot & \lambda_n \end{bmatrix} \quad (2.1.23)$$

where $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$ are the eigenvalues of the matrix \mathbf{A} .

Taylor Series

Consider the following function of n variables:

$$F(\mathbf{x}) = F(x_1, x_2, \dots, x_n) \quad (2.1.24)$$

The Taylor series expansion for this function, at the point \mathbf{x}^* is

$$F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^*)^T \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*) + \dots \quad (2.1.25)$$

where $\nabla F(\mathbf{x})$ is the gradient, and is defined as

$$\nabla F(\mathbf{x}) = \left[\frac{\partial}{\partial x_1} F(\mathbf{x}) \quad \frac{\partial}{\partial x_2} F(\mathbf{x}) \quad \dots \quad \frac{\partial}{\partial x_n} F(\mathbf{x}) \right]^T \quad (2.1.26)$$

and $\nabla^2 F(\mathbf{x})$ is the Hessian matrix, defined as

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial^2}{\partial x_1 \partial x_2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} F(\mathbf{x}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_2^2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} F(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} F(\mathbf{x}) & \frac{\partial^2}{\partial x_n \partial x_2} F(\mathbf{x}) & \cdots & \frac{\partial^2}{\partial x_n^2} F(\mathbf{x}) \end{bmatrix} \quad (2.1.27)$$

Minima [12][16]

Strong minimum: A point \mathbf{x}^* is a strong minimum of $F(\mathbf{x})$ if a scalar $\delta > 0$ exists, such that $F(\mathbf{x}) < F(\mathbf{x} + \Delta \mathbf{x})$ for all $\Delta \mathbf{x}$ such that $\delta > \|\Delta \mathbf{x}\| > 0$.

Weak minimum: A point \mathbf{x}^* is a weak minimum of $F(\mathbf{x})$ if it is not a strong minimum, and a scalar $\delta > 0$ exists, such that $F(\mathbf{x}) \leq F(\mathbf{x} + \Delta \mathbf{x})$ for all $\delta > \|\Delta \mathbf{x}\| > 0$.

If we move away from a strong minimum a small distance in any direction the function will increase.

The point \mathbf{x}^* is a unique global minimum of $F(\mathbf{x})$ if $F(\mathbf{x}) < F(\mathbf{x} + \Delta \mathbf{x})$ for all $\Delta \mathbf{x} \neq 0$.

For a strong minimum \mathbf{x}^* , the function may be smaller than the small neighborhood of \mathbf{x}^* . In such a case the strong minimum is called a local minimum. For a global minimum the function will be larger than the minimum point at any point in the domain.

Necessary and Sufficient conditions for Optimization

From the Taylor series we know that the first order necessary condition for \mathbf{x}^* to be a local minimum point for a function is the gradient at \mathbf{x}^* is equal to zero. i.e.

$$\nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*} = 0 \quad (2.1.28)$$

Any points that satisfy the above equation are called stationary points. Even though the above equation is satisfied, there is no guarantee that the local minimum is reached. The second order necessary condition for a strong minimum is that the Hessian matrix to be semidefinite. The sufficient conditions for a strong minimum to exist is the Hessian matrix to be positive definite. For a quadratic function

$$F(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{d}^T \mathbf{x} + c \quad (2.1.29)$$

we can decide if the function has a minimum or maximum by checking the eigenvalues of the Hessian matrix.

1. If the eigenvalues of the Hessian matrix are all positive, the function will have a strong minimum.
2. If the eigenvalues are all negative the function will have a strong maximum.
3. If some eigenvalues are positive and others are negative the function will have a saddle point.
4. If the eigenvalues are all nonnegative, but some eigenvalues are zero, the function will either have no stationary point or a weak minimum.
5. If the eigenvalues are all nonpositive, but some eigenvalues are zero, the function will either have a weak maximum or will have no stationary point.

We can consider that all analytic functions behave like quadratics over a small neighborhood.

All the optimization algorithms which we will discuss use iterative processes.[12][15] We begin from some initial guess, x_0 , and then update the guess according to the following equation

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)} \quad (2.1.30)$$

or

$$\Delta \mathbf{x}^{(k)} = (\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \alpha^{(k)} \mathbf{p}^{(k)} \quad (2.1.31)$$

2.2 Steepest Descent Method [12]

The objective of steepest decent is to satisfy the following condition:

$$F(\mathbf{x}^{(k+1)}) < F(\mathbf{x}^{(k)}) . \quad (2.2.1)$$

If the $\Delta \mathbf{x}^{(k)}$ is sufficient small, we can expand $F(\mathbf{x}^{(k+1)})$ as a first order Taylor series, i.e.

$$F(\mathbf{x}^{(k+1)}) = F(\mathbf{x}^{(k)}) + \mathbf{g}^{(k)\top} \Delta \mathbf{x}^{(k)} , \quad (2.2.2)$$

where $\mathbf{g}^{(k)\top}$ is the gradient evaluated at the point $\mathbf{x}^{(k)}$

$$\mathbf{g}^{(k)} = \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}} \quad (2.2.3)$$

In order for $F(\mathbf{x}^{(k+1)}) < F(\mathbf{x}^{(k)})$ to be satisfied, the second term of (2.2.2) must be negative. i.e.

$$\mathbf{g}^{(k)\top} \Delta \mathbf{x}^{(k)} = \alpha^{(k)} \mathbf{g}^{(k)\top} \mathbf{p}^{(k)} < 0 \quad (2.2.4)$$

We will select $\alpha^{(k)}$ (a.k.a learning rate in neural net publication) which is usually small, such that it is greater than zero. So

$$\mathbf{g}^{(k)\top} \mathbf{p}^{(k)} < 0$$

Any vector \mathbf{p}_k that satisfies (2.2.4) is called a descent direction. We need to find the steepest descent direction. Recalling the Schwarz inequality, we have

$$|\mathbf{p}^{(k)} \mathbf{g}^{(k)}| \leq \|\mathbf{p}^{(k)}\|_2 \|\mathbf{g}^{(k)}\|_2 \quad (2.2.5)$$

The equality holds if and only if $\mathbf{p}^{(k)} = \lambda \mathbf{g}^{(k)}$ where λ is a real number. Therefore if we select $\mathbf{p}^{(k)}$ such that

$$\mathbf{p}^{(k)} = -\mathbf{g}^{(k)} \quad (2.2.6)$$

then $|\mathbf{p}^{(k)} \mathbf{g}^{(k)}|$ has the maximum value which implies that the vector \mathbf{p}_k points to the steepest descent direction. The algorithm of steepest descent is as follows.

Algorithm 2.2.1 The algorithm for steepest descent

1. Set $k=0$, guess $\mathbf{x}^{(0)}$, select $\alpha^{(k)}$
2. Compute $\mathbf{g}^{(k)} = \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}}$
3. $\mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$
4. Compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$
5. If $\mathbf{x}^{(k+1)}$ satisfies the convergence criteria, then stop
6. Set $k=k+1$ goto 2

The learning rate $\alpha^{(k)}$ must be chosen to satisfy the following to guarantee convergence [16]

$$\alpha^{(k)} < \frac{2}{\lambda_{\max}} \quad (2.2.7)$$

Another method to choose the learning rate $\alpha^{(k)}$ is to minimize the performance function with respect to $\alpha^{(k)}$ at each iteration, i.e. we choose $\alpha^{(k)}$ to minimize

$$F(\mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}) \quad (2.2.8)$$

To minimize (2.2.8) we can take the derivative with respect to $\alpha^{(k)}$ and set it to zero.

$$\frac{d}{d\alpha^{(k)}} F(\mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}) = \nabla F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^{(k)}} \mathbf{p}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)T} \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}} \mathbf{p}^{(k)} = 0 \quad (2.2.9)$$

We can solve for $\alpha^{(k)}$

$$\alpha^{(k)} = -\frac{\nabla F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^{(k)}} \mathbf{p}^{(k)}}{\mathbf{p}^{(k)T} \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}} \mathbf{p}^{(k)}} = -\frac{\mathbf{g}^{(k)T} \mathbf{p}^{(k)}}{\mathbf{p}^{(k)T} \mathbf{H}^{(k)} \mathbf{p}^{(k)}} \quad (2.2.10)$$

where $\mathbf{H}^{(k)}$ is the Hessian matrix evaluated at point $\mathbf{x}^{(k)}$, i.e.

$$\mathbf{H}^{(k)} = \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}}$$

This method is also called a line search algorithm.

Algorithm 2.2.2 The algorithm for the steepest descent with line search

1. Set $k=0$, guess $\mathbf{x}^{(k)}$
2. $\mathbf{g}^{(k)} = \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}}$
3. $\mathbf{p}^{(k)} = -\mathbf{g}^{(k)}$
4. Compute $\alpha^{(k)} = -\frac{\nabla F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^{(k)}} \mathbf{p}^{(k)}}{\mathbf{p}^{(k)T} \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}} \mathbf{p}^{(k)}} = -\frac{\mathbf{g}^{(k)T} \mathbf{p}^{(k)}}{\mathbf{p}^{(k)T} \mathbf{H}^{(k)} \mathbf{p}^{(k)}}$
5. Compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$
6. If $\mathbf{x}^{(k)}$ satisfies the convergence criteria, then stop
7. Set $k=k+1$ goto 2

The advantage of steepest descent is that it is simple and will converge as long as $\alpha^{(k)}$ satisfies (2.2.7). However because the method is based on the first-order Taylor series, the convergence rate is often very slow. Also steepest descent is not scale-invariant. If we replace one component x_i by $C \cdot x_i$, the speed of convergence may be greatly changed.

2.3 Newton's Method

Newton's method is based on the second-order Taylor's series [12].

$$F(\mathbf{x}^{(k+1)}) = F(\mathbf{x}^{(k)} + \Delta \mathbf{x}) = F(\mathbf{x}^{(k)}) + \mathbf{g}^{(k)\top} \Delta \mathbf{x}^{(k)} + \frac{1}{2} \Delta \mathbf{x}^{(k)\top} \mathbf{H}^{(k)} \Delta \mathbf{x}^{(k)} \quad (2.3.1)$$

Taking the derivative with respect to $\Delta \mathbf{x}^{(k)}$ and setting it to zero, we have

$$\mathbf{g}^{(k)} + \mathbf{H}^{(k)} \Delta \mathbf{x}^{(k)} = 0 \quad (2.3.2)$$

Solving for $\Delta \mathbf{x}^{(k)}$, we have

$$\Delta \mathbf{x}^{(k)} = -\mathbf{H}^{(k)-1} \mathbf{g}^{(k)} \quad (2.3.3)$$

So Newton's method can be represented by

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{H}^{(k)-1} \mathbf{g}^{(k)} \quad (2.3.4)$$

where $\mathbf{H}^{(k)}$ is the Hessian matrix evaluated at point \mathbf{x}_k . i.e.

$$\mathbf{H}^{(k)} = \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}} \quad (2.3.5)$$

In practice the inverse matrix is not computed as this is too slow.

Algorithm 2.3.1: The algorithm for Newton's method is

1. Set $k=0$, guess $\mathbf{x}^{(0)}$
2. Compute $\mathbf{g}^{(k)} = \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}}$ and $\mathbf{H}^{(k)} = \nabla^2 F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}}$,

3. Compute $\Delta \mathbf{x}^{(k)}$ by solving the following equations

$$\mathbf{H}^{(k)} \Delta \mathbf{x}^{(k)} = -\mathbf{g}^{(k)}$$

4. Compute $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)}$

5. If $\mathbf{x}^{(k)}$ satisfies the convergence criteria, then stop

6. Set $k=k+1$ goto 2

It can be shown that the rate of convergence of Newton's method is second-order if Hessian matrix is positive definite. If the function is quadratic, Newton's method will converge in one step. Quadratic convergence is the fastest rate normally encountered in nonlinear optimization and for this reason Newton's method is of fundamental importance. However, very few practical problems have a Hessian matrix that is everywhere positive definite. Even if the Hessian matrix $\mathbf{H}^{(k)}$ is positive definite at a nonstationary point, $\mathbf{x}^{(k+1)}$ may lie outside the region where the quadratic approximation at $\mathbf{x}^{(k)}$ is valid. This can be a problem when the curvature of the function in part of the region between $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(k)} + \mathbf{p}^{(k)}$ is sharper than that predicted by second derivatives alone. In this case $\alpha^{(k)} = 1$ can be too big a step because it is possible for the function to increase again. An improvement that could overcome this is to determine $\alpha^{(k)}$ by linear search. However such search is undesirable [12] because it slows down the method substantially.

There are three more serious difficulties. The first is the possibility that $\mathbf{g}^{(k)T} \Delta \mathbf{x}^{(k)} = 0$ when $\mathbf{g}^{(k)} \neq 0$, in which case $\mathbf{x}^{(k)}$ is already the minimum along $\Delta \mathbf{x}^{(k)}$ and no further progress is possible. The second difficulty is that $\mathbf{H}^{(k)}$ may be singular, in

which case there is either no solution to (2.3.2) or else there are infinite number of solutions. Finally, if $\mathbf{x}^{(k)}$ is a saddle point at which $\mathbf{H}^{(k)}$ is non-singular, then $\mathbf{g}^{(k)} = 0$ and (2.3.2) can be satisfied only if $\mathbf{p}^{(k)} = 0$ which is obviously useless as a search vector. Clearly, Newton's method is not a satisfactory general-purpose algorithm for function minimization. Fortunately it can be modified to provide reliable algorithms. The general philosophy behind these modified Newton's method is to replace the Hessian matrix with a matrix that is guaranteed always to be positive definite and which is otherwise close to Hessian matrix. For a special form of the performance function such as least squares, Gauss-Newton method [12][14] and the Levenberg-Marquardt [28][29] method are very efficient alternatives to Newton's method.

2.4 The Conjugate Directions Method [12][14][15]

The Newton method has the advantage of requiring only one iteration to converge on a quadratic function which is one form of quadratic termination. However it requests to calculate and store the second derivatives of the Hessian matrix. The conjugate direction method is to search the minimum in the conjugate direction to guarantee quadratic termination. Suppose that we want to minimize the function (2.1.29). We define the conjugate directions as follows:

Definition 2.4.1:

A set of vectors $\{\mathbf{p}_k\}$ is mutually conjugate with respect to a positive definite Hessian matrix \mathbf{A} if and only if

$$\mathbf{p}_k^T \mathbf{A} \mathbf{p}_j = 0 \quad k \neq j \quad (2.4.1)$$

There are a lot of vectors that satisfies (2.4.1). One set consists of the eigenvectors of A.

It can be shown [12] that if we make a sequence of exact linear searches along any set of conjugate directions $\{p_1, p_2, \dots, p_n\}$, then the exact minimum of any quadratic function with n parameter, will be reached in at most one cycle of n searches. Recall that for quadratic function, the gradient is

$$\nabla F(\mathbf{x}) = \mathbf{Ax} + \mathbf{d} \quad (2.4.2)$$

If we calculate the change in the gradient at iteration k+1, we have

$$\Delta \mathbf{g}^{(k)} = \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)} = (\mathbf{Ax}^{(k+1)} + \mathbf{d}) - (\mathbf{Ax}^{(k)} + \mathbf{d}) = \mathbf{A}\Delta \mathbf{x}^{(k)} \quad (2.4.3)$$

From equation (2.2.2), we have

$$\Delta \mathbf{x}^{(k)} = (\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \alpha^{(k)} \mathbf{p}^{(k)} \quad (2.4.4)$$

where $\alpha^{(k)}$ is chosen to minimize F(x) in the direction $\mathbf{p}^{(k)}$.

We can now restate the conjugate conditions by substituting (2.4.2) and (2.4.3) into (2.4.1).

$$\alpha^{(k)} \mathbf{p}^{(k)\top} \mathbf{A} \mathbf{p}^{(j)} = \Delta \mathbf{x}^{(k)\top} \mathbf{A} \mathbf{p}^{(j)} = \Delta \mathbf{g}^{(k)\top} \mathbf{p}^{(j)} = 0 \quad \mathbf{k} \neq \mathbf{j} \quad (2.4.5)$$

Usually we use the steepest descent method to begin the search, i.e.

$$\mathbf{p}^{(0)} = -\mathbf{g}^{(0)} \quad (2.4.6)$$

Then at each iteration we need to construct a vector $\mathbf{p}^{(k)}$ which is orthogonal to $\{\Delta \mathbf{g}^{(0)}, \Delta \mathbf{g}^{(1)}, \dots, \Delta \mathbf{g}^{(k-1)}\}$. We can use Gram-Schmidt orthogonalization (2.1.18). It can be simplified [12] to the following form

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

The $\beta^{(k)}$ can be chosen by several different methods, which will produce equivalent results for quadratic functions. The most common choices [12] are

$$\beta^{(k)} = \frac{\Delta\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k)}}{\Delta\mathbf{g}^{(k-1)\top}\mathbf{p}^{(k)}} \quad (2.4.8)$$

developed by Hestenes and Stiefel,

$$\beta^{(k)} = \frac{\mathbf{g}^{(k)\top}\mathbf{g}^{(k)}}{\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k-1)}} \quad (2.4.9)$$

developed by Fletcher and Reeves, and

$$\beta^{(k)} = \frac{\Delta\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k)}}{\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k-1)}} \quad (2.4.10)$$

developed by Polak and Ribiere.

The algorithm is as follows:

Algorithm 2.4.1: The conjugate gradient method

1. Set $k=0$, guess $\mathbf{x}^{(0)}$
2. Select the first search direction according to the steepest descent method, i.e.

$$\mathbf{p}^{(0)} = -\mathbf{g}^{(0)}$$

3. Calculate $\mathbf{g}^{(k)}$ according to (2.2.5), i.e.

$$\mathbf{g}^{(k)} = \nabla F(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(k)}}$$

4. Calculate the $\beta^{(k)}$ according to (2.4.8) or (2.4.9) or (2.4.10).

5. Calculate $\mathbf{p}^{(k)}$ according to (2.4.7), i.e.

$$\mathbf{p}^{(k)} = -\mathbf{g}^{(k)} + \beta^{(k)}\mathbf{p}^{(k-1)}$$

6. Calculate $\Delta \mathbf{x}^{(k)}$ according to (2.4.4), i.e.

$$\Delta \mathbf{x}^{(k)} = (\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \alpha^{(k)} \mathbf{p}^{(k)}$$

Choosing $\alpha^{(k)}$ to minimize $F(\mathbf{x})$ along $\mathbf{x} = \mathbf{x} + \alpha^{(k)} \mathbf{p}^{(k)}$

7. Calculate $\mathbf{x}^{(k+1)}$ as follows

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)}$$

8. If $\mathbf{x}^{(k+1)}$ satisfies the convergence criteria, stop

9. Goto step 3.

3. ARTIFICIAL NEURAL NETWORK LEARNING ALGORITHMS

In the previous chapter, we introduced some basic optimization theory. Now we will apply the theory to artificial neural networks. In particular, we will describe the architecture, dynamic adjustment, computation and conjugate gradient learning algorithms in artificial neural networks.

3.1 Architectures of Feedforward Artificial Neural Networks

In chapter one, we know that there are basically three types of artificial neural networks. This thesis will focus on the most widely used type, multilayer feedforward networks. The architecture of a multilayer feedforward network is shown in Figure 3.1.1. Such a network arranges neurons in layers. All neurons in a layer are connected to all neurons in the adjacent layers through unidirectional links. These links are represented by synaptic weights. Notice that we treat the input layer of the network as some connection nodes. The hidden layers of the network also consists of some connection nodes. The hidden layers of a network are all of the layers except the input and output layers of the network. So the number of hidden layers is the number of layers in a network minus one. Generally speaking there is no theoretical limit on the number of hidden layers, but in practice one or two hidden layers is usually enough to model even the most complex problems. It has been shown that it is sufficient to use a maximum of three layers (two hidden layers and one output layer) to solve an arbitrarily complex pattern recognition problems [15].

The notations we will use are shown in Figure 3.1.1. All neurons in a layer are consecutively indexed beginning from 1, in an top - down fashion. The layers are indexed in a left-to-right order and are identified by square-bracketed superscripts. All inputs to a

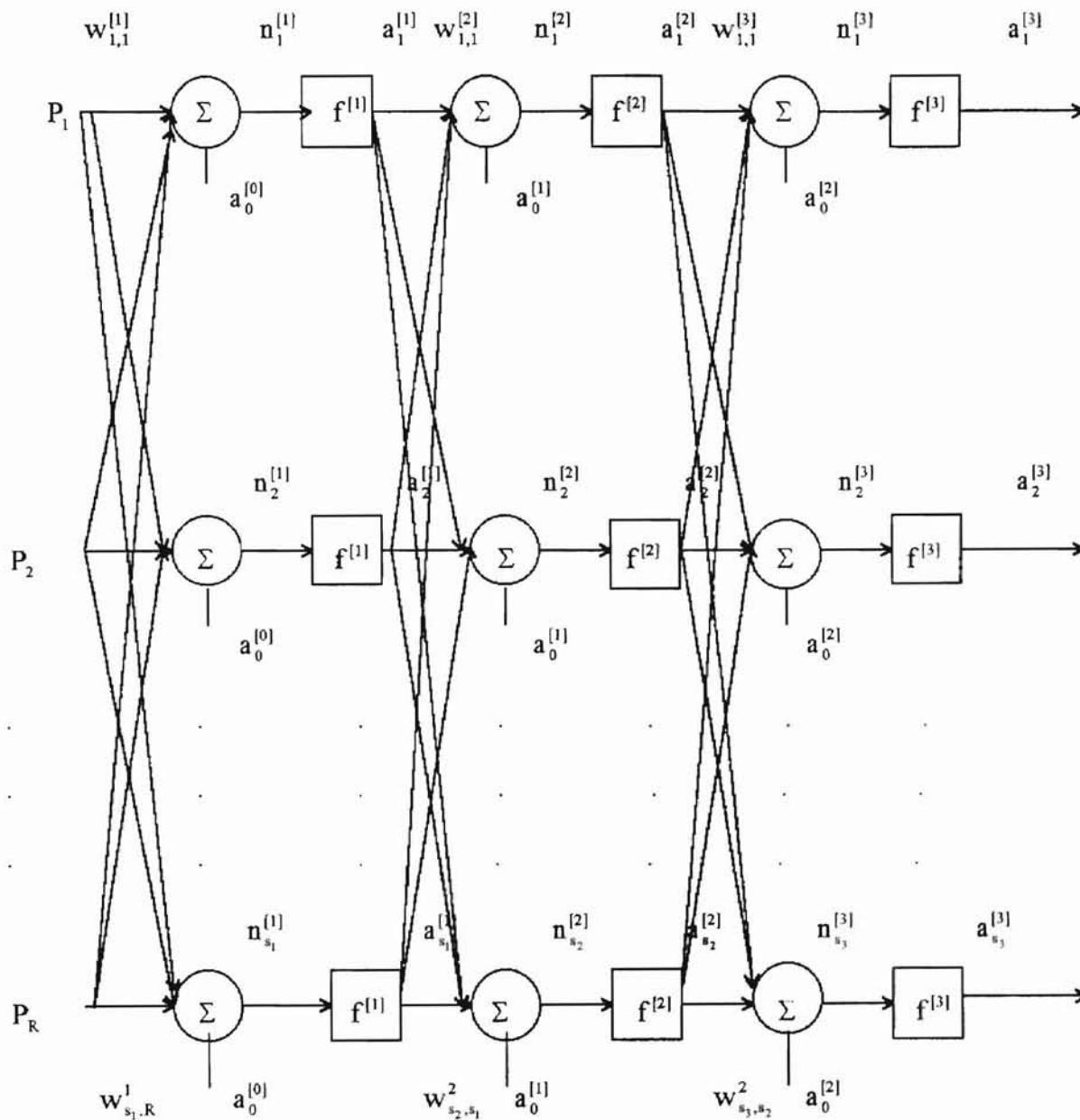


Figure 3.1.1. A Three-Layer Feedforward Network

neuron in layer k are denoted as $a_i^{[k-1]}$ where $i = 0, 1, 2, \dots, S_{k-1}$ (S_{k-1} is the number of neurons in $(k-1)$ th layer). In the case of $k-1 = 0$, $a_i^{[0]}$ are the inputs of the network. For each layer, we assumed an extra bias node which has a constant output value of -1, i.e., $a_0^{[k]} = -1$ for all $k = 0, 1, \dots, K-1$. Notice that for each $k > 2$, $a_i^{[k-1]}$ is also the output of neuron i in $(k-1)$ th layer. The outputs in the k th layer of network can be written in vector form as $\mathbf{a}^{[k]}$. A weight is represented as $w_{ji}^{[k]}$, $j \neq 0$ where k is the layer index and "j,i" means that the weight is the connection from the i th neuron in layer $k-1$ to the j th neuron in layer k . In vector form, weights can be represented by $\mathbf{w}^{[k]} = \left(w_{ji}^{[k]} \right)^T$. The $n_j^{[k]}$ represents the weighted sum of a neuron j in layer k . The weighted sum of the inputs of a neuron j in layer k can be expressed as

$$n_j^{[k]} = \sum_{i=0}^{S_k} w_{ji}^{[k]} \cdot a_i^{[k-1]} \tag{3.1.1}$$

The output of the neuron j in layer k can be expressed as

$$a_j^{[k]} = f^{[k]} \left(n_j^{[k]} \right) \quad j = 1, 2, \dots, n_k \tag{3.1.2}$$

where $f_j^{[k]}$ is the activation function of the neuron.

In vector form, there formula can be written as

$$\mathbf{n}^{[k]} = \left(\mathbf{w}^{[k]} \right)^T \mathbf{a}^{[k-1]} \tag{3.1.3}$$

$$\mathbf{a}^{[k]} = \mathbf{f}^{[k]} \left(\mathbf{n}^{[k]} \right) \tag{3.1.4}$$

where $\mathbf{f}^{[k]} = \left(f^{[k]} \right)^T$ is a vector of activation function values.

The original activation function was a binary (hard-limiting) function [3]. This limits the application of perceptron neural networks to only classification problem. In order to solve a general type of mapping application problems, we need to use nonlinear continuous activation functions. There are many nonlinear activation functions that can be used in multilayer networks as long as the functions are differentiable. The most commonly used functions are the sigmoid function and the hyperbolic function which are expressed as follows:

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

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

The graphs of the sigmoid and hyperbolic functions are shown in Figure 3.1.2 and Figure 3.1.3. Since we can always scale down the input and output values to the interval (0, 1) or (-1, 1), there is no significant difference between the two functions. In this paper, the sigmoid function is used.

The weights in a neural network are initially chosen to be small random numbers. Since the activation function is active only in a small domain interval as shown in Figure 3.1.2, we should choose the initial weights to be small values. If the initial weights are too large, the activation functions may saturate at the beginning of the training and the network is prone to get stuck in a local minimum near the starting point [16]. In this paper, the initial weights of all neural networks are chosen as random numbers uniformly distributed between $\frac{-0.5}{\text{fan-in of that node}}$ and $\frac{0.5}{\text{fan-in of that node}}$ [15], where the fan-in of that node is the number of inputs including bias that are input to that node.

Forward computations

As we know from chapter 1, a neural network learning process includes two phases: forward computation and backward computation. During the forward computation, a set of input data is given to the neurons in the first layer (input layer). These neurons are activated and pass the results to neurons in the next layer. The process continues until the output layer is reached and the outputs of the network have been calculated. The process can be summarized as follows:

1. Given input vector \mathbf{x} , set $\mathbf{n}^{[0]} = \mathbf{x}$
2. The weight matrix and activation function $\mathbf{f}^{[k]}$, $k = 1, 2, \dots, M$ are known, where M is the number of layers in the network.
3. Compute $\mathbf{n}^{[k]} = (\mathbf{w}^{[k]})^T \mathbf{a}^{[k-1]}$ and $\mathbf{a}^{[k]} = \mathbf{f}^{[k]}(\mathbf{n}^{[k]})$ for $k=1, 2, \dots, M$.
4. $\mathbf{a}^{[M]}$ is the output of the network.

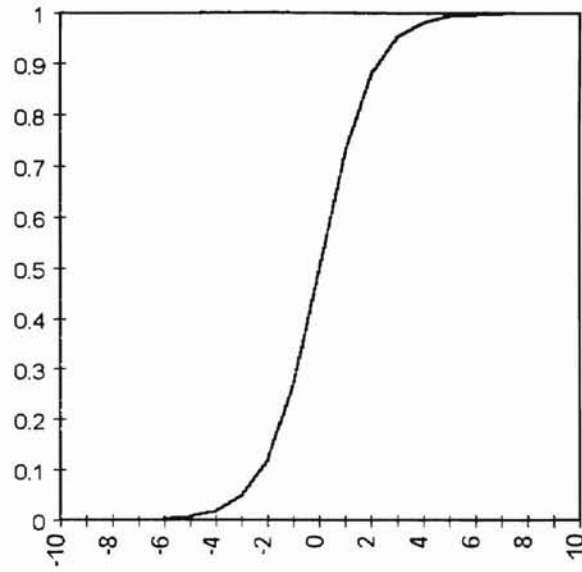


Figure 3.1.2 The Sigmoid Function

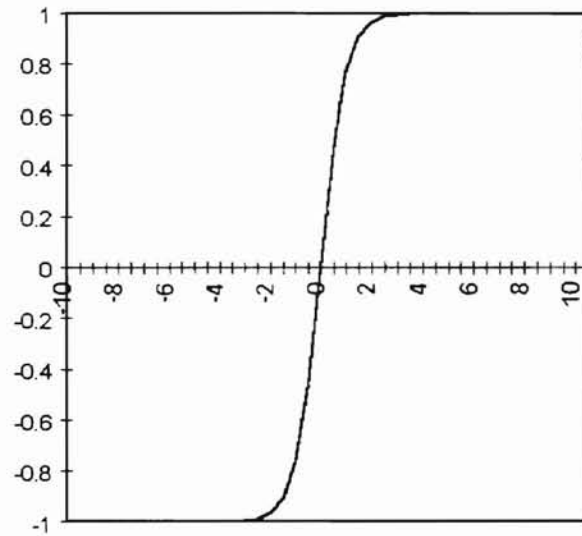


Figure 3.1.3 The Hyperbolic Function

3.2 Dynamic Behavior in Feedforward Artificial Neural Network

A feedforward artificial neural network changes its behavior (weights) dynamically during the training session. The error made by the network during training is measured by a predefined function called the error function (performance) function [15], cost function [24] or energy function [26]. The error function is used to calculate the errors and the distribution of errors among all neurons of a network. Then the connection weights are changed to reduce the error of the network. This dynamic adaptation of weights ends when the error is within a tolerance limit or an optimum point has been reached with respect to some optimization criterion. We will discuss generalization in the next section. Now we explain in detail some concepts involved in an artificial neural network training process.

As discussed in chapter one, learning can be divided into supervised learning and unsupervised learning. Supervised learning is used in this thesis. Supervised learning implies a situation in which the network is functioning as an input/output system. In other words, the network receives an input vector and calculates an output vector using forward calculation. This output vector is compared with the “desired” or “correct” output vector. The error is backpropagated through the network to adjust the weights until the error is small and generalization is acceptable. Normally we need two sets of input/output data. One is for training purpose, the other is used to test the network after it has been trained. The number of input/output data vectors in the training set depends on the number of weights in the network. A general rule of thumb is that the number of data vector in training set must be much larger than the number of parameters (weights) to avoid overfitting [2][23]. Overfitting will be discussed in the next section.

There are two methods used to adjust weights during training process. One is called on-line learning. The other is called off-line learning (also called batch learning). In on-line learning, weights are adjusted each time an input is presented to a neural network and errors have been produced. In off-line learning, weight updating is deferred until all inputs have been presented to the network. The comparison of on-line and off-line learning is listed as follows [15].

1. On-line learning is usually convenient and more effective than off-line learning when the number of training examples is very large.
2. On-line learning introduces some randomness (noise) that often may help escape from local minima.
3. Usually, On-line learning is faster and more effective than off-line learning, especially for large-scale classification problems.
4. However, for many applications, especially if high precision mapping is required, off-line learning may be the method of choice.
5. Off-line learning lends itself to straightforward application of more sophisticated optimization procedures.

Practically, the relative effectiveness of on-line and off-line learning is highly dependent on the problem. From the optimization point of view, off-line learning is more suitable to implement learning algorithms.

3.3 Overfitting and Generalization in Artificial Neural Networks

We have introduced basic concept of overfitting in chapter one. Now we explain some mechanisms behind this phenomenon.

When a network is trained, the weights are modified in order to decrease the errors in the training data set. If the network is tested on a new set of data, initially the errors in the test data set tend to decrease in step with the training error as the network tries to generalize from the training data set. However if the training data are incomplete, it may contain spurious and misleading regularities due to sampling [2][23]. Therefore as training continues, the errors in the test data set increase. Figure 3.3.1 illustrates this situation schematically.

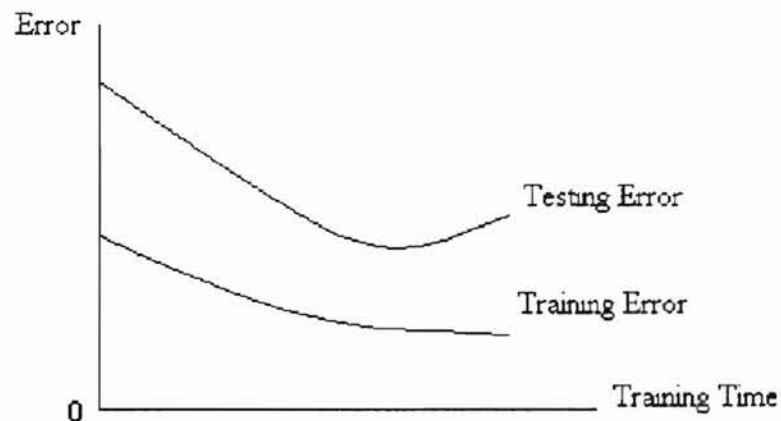


Figure 3.3.1 The Relationship Between Training Error and Testing Error

Mathematically, the objective of learning in the neural network is to infer a function from a given sample data set. Learning algorithms are essentially to search for a function that fits the given data in the specified space of functions. After learning, the neural network is able to maximize its predictive accuracy in the new data set. If we work too hard to find the best fit to the training data, there is a risk that we will fit the noise in the data by memorizing various peculiarities of the training data rather than finding a

general predictive rule [40]. It is generally agreed that overfitting is closely related to the architecture of the network, i.e., the size of network. If training starts with too small a network for the problem, no learning can occur. If the network is too large, it may be vulnerable to overfitting [44]. The question is what size network gives valid generalization. Eric B. Baum and David Haussler [33] analyzed theoretically the lower and upper bounds on the size of the sample size vs. network size needed to achieve valid generalization. Their conclusion is as follows:

Given m random training examples chosen from an arbitrary probability distribution, assume

$0 \leq \varepsilon \leq 1/8$ (ε is called the accuracy parameter), it can be proved that if

$m \geq O\left(\frac{W}{\varepsilon} \log \frac{N}{\varepsilon}\right)$ random examples can be loaded on a feedforward network, so that at

least a fraction $1 - \frac{\varepsilon}{2}$ of the examples are correctly classified, then one has confidence

approaching certainty that the network will correctly classify a fraction $1 - \varepsilon$ of future test examples drawn from the same distribution. The lower bound for the number of

random examples is $\Omega\left(\frac{W}{\varepsilon}\right)$. Although these results are very encouraging, the theoretical

bounds are quite crude and the gap between the upper and lower bound on the worst case

sample size for architectures with one hidden layer remains open. Also, the case of

multiple hidden layers is still open. Finally, the result applies only to the threshold

functions, although these authors conjectured it might apply to nonlinear functions such as

sigmoid as well.

Subutai Ahmad and Gerald Tasauro [35] analyzed how many training patterns and training cycles are needed for a problem of a given size and difficulty, how to represent the input, and how to choose training examples. They concluded that the performance of a network is closely related to the number of training patterns and the size of the network. Their results showed that for a fixed network size, the failure rate decreases exponentially with the size of the training set. The number of patterns requires to achieve a fixed performance level was shown to increase linearly with the network size.

To summarize, overfitting is related to the degrees of freedom of a neural network. The degrees of freedom of a neural network includes not only the weights but also the potential non-linearity of the network, the architecture, and the number of data vectors used during training [26].

3.4 Stopped Training Method to Reduce Overfitting

Having discussed some mechanisms and factors that affect overfitting, we are ready to explore methods to reduce overfitting. There are many methods to reduce overfitting and improve generalization [23]. Two categories that are widely used are the stopped training method [23][30][36] and penalty method [23][24][26]. We will explain the stopped training method in this section and penalty methods in the next section.

The stopped training method estimates the generalization ability during training and stops when the generalization ability begins to decrease (i.e. the testing error begins to increase). Experimental experience suggests that the training and generalization behavior in Figure 3.3.1 is typical [2][23]. In order to find the minimum of the test error, we divide the data into a training set and validation set. At periodic intervals, the process of network

training is stopped temporarily, the weights are temporarily frozen and the network generalization is tested by the validation set using mean squared error. The mathematical foundation for this method is the cross-validation method of statistics [46].

3.5 Penalty Method to Reduce Overfitting

Although the stopped training method is straightforward, it may not be practical when only a limited amount of data is available. Another way to reduce overfitting is to use a penalty method [23][24][26][34][38]. The basic approach involves adding penalty terms to the usual error function in order to constrain the search and cause weights to decay differentially. (So a penalty method is also called a constrained optimization method). This is very similar to many proposals in statistical regression where a “simplicity” measure is minimized along with the error term and is sometimes referred to as ridge regression and biased regression [41]-[44]. Basically, the statistical concept of biased regression derives from parameter estimation approaches that attempt to achieve a best linear unbiased estimator (called “BLUE”). By definition an unbiased estimator is one with the lowest possible variance and theoretically, unless there is significant collinearity or nonlinearity among the variables, a least squares estimator(LSE) can be shown to be a BLUE. However if input variables are correlated or nonlinear with the output (as in the case in back-propagation) then there is no guarantee that the LSE will also be unbiased. Consequently, introducing a bias (penalty) term may actually reduce the variance of the estimator below that of the theoretically unbiased estimator.

Now the question is what types of penalty term shall we choose. There are many different types of penalty term used in neural networks to reduce overfitting [23]. Some of

them have a disadvantage in that large weights decay at the same rate as small weights. It is possible to design biases that influence weights when they are relatively small or even in a particular range of values [37]. One form used in this thesis is a rectangular hyperbolic function defined as follows:

$$f(w) = \frac{w^2}{1 + w^2} \quad (3.5.1)$$

After taking the derivative with respect to w , we have the following first derivative of $f(w)$:

$$f'(w) = -\frac{2w}{1 + w^2} \quad (3.5.2)$$

The derivative of $f(w)$ is plotted in Figure 3.5.1. It is non-monotonic showing a strong differential effect on small weights close to the origin (+ or -). It approximates to zero when the weights are far away from the origin which means it has little effect on large weights. The object is to reduce weights that are small and unimportant to values very close to zero. After that, these connections could be removed from the network. Any neurons that became disconnected during this pruning process could be removed. This results in a simple and more parsimonious neural model of the problem.

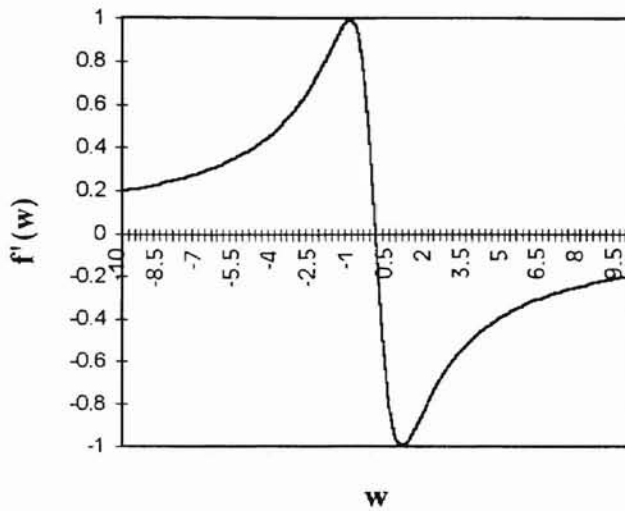


Figure 3.5.1 The First Derivative of the Penalty Term

3.6 Computation in Feedforward Artificial Neural Networks

We have discussed forward computation in feedforward artificial neural networks. Now we will formulate the backpropagation computation in feedforward artificial networks. Considering a neural network of M layer, the performance function is defined as follows:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^Q \left((f^{[k]}(\mathbf{p}_i, \mathbf{w}) - \mathbf{t}_i)^T (f^{[k]}(\mathbf{p}_i, \mathbf{w}) - \mathbf{t}_i) + \lambda \sum_1 \frac{\mathbf{w}_1^2 / \mathbf{w}_0^2}{1 + \mathbf{w}_1^2 / \mathbf{w}_0^2} \right) \quad (3.6.1)$$

The first term is the performance function (error function). The second term is the penalty term. It sums over all connection weights. Q is the number of input/output samples. \mathbf{p}_i is the i th input datum. \mathbf{t}_i is the desired i th output. λ and \mathbf{w}_0 are constants that are adjusted during training. Because the differentiation is additive, it is convenient to consider one input/output sample i . In practice, this is used for on-line training. Summation over the entire set of input/output samples constitutes off-line training. So we have

$$E_i = \frac{1}{2} \left((\mathbf{f}^{[k]}(\mathbf{p}_i, \mathbf{w}) - \mathbf{t}_i)^T (\mathbf{f}^{[k]}(\mathbf{p}_i, \mathbf{w}) - \mathbf{t}_i) + \lambda \sum_i \frac{\mathbf{w}_i^2 / \mathbf{w}_0^2}{1 + \mathbf{w}_i^2 / \mathbf{w}_0^2} \right) \quad (3.6.2)$$

To calculate the gradient element g_{ji} , we take the derivative of E_i with respect to $w_{ji}^{[k]}$ and, using chain rule, we have

$$g_{ji}^{[k]} = \frac{\partial E_i}{\partial w_{ji}^{[k]}} = \frac{\partial E_i}{\partial a_j^{[k]}} \cdot \frac{\partial a_j^{[k]}}{\partial w_{ji}^{[k]}} + p_{ji} \quad (3.6.3)$$

where p_{ji} is an element of penalty term and is defined as

$$p_{ji} = \lambda \frac{w_{ji}^{[k]} w_0^2}{\left((w_{ji}^{[k]})^2 + w_0^2 \right)} \quad (3.6.4)$$

From (3.1.1), we have

$$\frac{\partial a_j^{[k]}}{\partial w_{ji}^{[k]}} = a_i^{[k-1]} \quad (3.6.5)$$

If we define [16]

$$s_j^{[k]} \equiv \frac{\partial E_i}{\partial w_{ji}^{[k]}} = s_j^{[k]} \cdot a_i^{[k-1]} + p_{ji} \quad (3.6.6)$$

(s_j is called the sensitivity of E_i to change in the j th element of the net input at layer k),

then (3.6.3) becomes

$$g_{ji}^{[k]} = \frac{\partial E_i}{\partial w_{ji}^{[k]}} = s_j^{[k]} \cdot a_i^{[k-1]} + p_{ji} \quad (3.6.7)$$

To derive the recurrence relationship for the sensitivities, we will use the Jacobian matrix which we have already introduced in Chapter 2.

$$\frac{\hat{\mathbf{a}}^{[k+1]}}{\hat{\mathbf{a}}^{[k]}} \equiv \begin{bmatrix} \frac{\hat{a}_1^{[k+1]}}{\hat{a}_1^{[k]}} & \frac{\hat{a}_1^{[k+1]}}{\hat{a}_2^{[k]}} & \dots & \dots & \frac{\hat{a}_1^{[k+1]}}{\hat{a}_{s_k}^{[k]}} \\ \frac{\hat{a}_2^{[k+1]}}{\hat{a}_1^{[k]}} & \frac{\hat{a}_2^{[k+1]}}{\hat{a}_2^{[k]}} & \dots & \dots & \frac{\hat{a}_2^{[k+1]}}{\hat{a}_{s_k}^{[k]}} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \frac{\hat{a}_{s_{k+1}}^{[k+1]}}{\hat{a}_1^{[k]}} & \frac{\hat{a}_{s_{k+1}}^{[k+1]}}{\hat{a}_2^{[k]}} & \dots & \dots & \frac{\hat{a}_{s_{k+1}}^{[k+1]}}{\hat{a}_{s_k}^{[k]}} \end{bmatrix} \quad (3.6.8)$$

Now consider the element ij in (3.6.8), we have

$$\begin{aligned} \frac{\hat{a}_i^{[k+1]}}{\hat{a}_j^{[k]}} &= \frac{\partial \left(\sum_{l=1}^{s_k} w_{il}^{[k+1]} a_l^{[k]} \right)}{\hat{a}_j^{[k]}} = w_{ji}^{[k+1]} \frac{\hat{a}_j^{[k]}}{\hat{a}_j^{[k]}} \\ &= w_{ij}^{[k+1]} \frac{\mathcal{A}^{[k]}(\mathbf{n}_j^{[k]})}{\hat{a}_j^{[k]}} = w_{ij}^{[k+1]} \mathbf{f}^{[k]}(\mathbf{n}_j^{[k]}) \end{aligned} \quad (3.6.9)$$

where

$$\mathbf{f}^{[k]}(\mathbf{n}_j^{[k]}) = \frac{\mathcal{A}^{[k]}(\mathbf{n}_j^{[k]})}{\hat{a}_j^{[k]}} \quad (3.6.10)$$

So the Jacobian matrix can be written as

$$\frac{\hat{\mathbf{a}}^{[k+1]}}{\hat{\mathbf{a}}^{[k]}} = \mathbf{W}^{[k+1]} \cdot \mathbf{F}(\mathbf{n}^{[k]}) \quad (3.6.11)$$

where

$$\dot{\mathbf{F}}^{[k]}(\mathbf{n}^{[k]}) = \begin{bmatrix} \dot{f}^{[k]}(\mathbf{n}_1^{[k]}) & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & \dot{f}^{[k]}(\mathbf{n}_2^{[k]}) & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot & \cdot & \dot{f}^{[k]}(\mathbf{n}_{S_k}^{[k]}) \end{bmatrix} \quad (3.6.12)$$

We can now write the sensitivity recursively in matrix form as follows

$$\begin{aligned} \mathbf{s}^{[k]} &= \frac{\partial \mathcal{E}_i}{\partial \mathbf{n}^{[k]}} = \left(\frac{\partial \mathbf{n}^{[k+1]}}{\partial \mathbf{n}^{[k]}} \right)^T \frac{\partial \mathcal{E}_i}{\partial \mathbf{n}^{[k+1]}} = \dot{\mathbf{F}}(\mathbf{n}^{[k]}) \cdot (\mathbf{W}^{[k+1]})^T \cdot \frac{\partial \mathcal{E}_i}{\partial \mathbf{n}^{[k+1]}} \\ &= \dot{\mathbf{F}}(\mathbf{n}^{[k]}) \cdot (\mathbf{W}^{[k+1]})^T \cdot \mathbf{s}^{[k+1]} \end{aligned} \quad (3.6.13)$$

Now we can see the recurrent relationship of the sensitivity. The sensitivities are propagated backward through the network from the last layer to the first layer. In order to complete the backpropagation, we need to know the starting point of the backpropagation. The starting point can be obtained from the output layer.

$$\mathbf{s}_i^{[K]} = \frac{\partial \mathcal{E}_i}{\partial \mathbf{n}_i^{[K]}} = -(t_i - a_i) \frac{\partial a_i}{\partial \mathbf{n}_i^{[K]}} \quad (3.6.14)$$

Since

$$\frac{\partial a_i}{\partial \mathbf{n}_i^{[K]}} = \frac{\partial a_i^{[K]}}{\partial \mathbf{n}_i^{[K]}} = \dot{f}^{[K]}(\mathbf{n}_i^{[K]}) \quad (3.6.15)$$

we can write

$$\mathbf{s}_i^{[K]} = -(t_i - a_i) \dot{f}^{[K]}(\mathbf{n}_i^{[K]}) \quad (3.6.16)$$

In matrix form (3.6.15) can be expressed as

$$\mathbf{s}^{[K]} = -\dot{\mathbf{F}}^{[K]}(\mathbf{n}^{[K]})(\mathbf{t} - \mathbf{a}) \quad (3.6.17)$$

So we can recursively calculate the sensitivities from the last layer to the first layer. Knowing the sensitivities, we can calculate the gradient according to (3.6.6). The following algorithm is the off-line model based on Algorithm 2.4.1

Algorithm 3.6.1: Given a set of $S = \{(\mathbf{q}_i, \mathbf{t}_i) | \mathbf{q}_i \text{ is input, } \mathbf{t}_i \text{ is desired output of } \mathbf{q}_i\}$

of d training samples and given a network of K layers with input dimension u and output dimension v .

1. Initialize all weights $\mathbf{w}^{[kl]} = (w_{ji}^{[kl]})$, $l = 1, 2, \dots, K$ as random numbers uniformly distributed between $\frac{-0.5}{\text{fan-in of that unit}}$ and

$\frac{0.5}{\text{fan-in of that unit}}$. Set w_0, λ .

Initialize $\mathbf{g}^{(k)} = 0$.

2. For each sample $(\mathbf{x}_i, \mathbf{t}_i) \in S$, repeat the following steps.

2.1 Compute the actual outputs of network according to (3.1.3)

and (3.1.4) using the weight $\mathbf{w}^{(k)}$

2.2 Calculate the gradient $\mathbf{g}(\mathbf{x}_i)$ according to (3.6.3)

2.3 Sum up $\mathbf{g}(\mathbf{x}_i)$, i.e., $\mathbf{g}^{(k)} = \mathbf{g}^{(k)} + \mathbf{g}(\mathbf{x}_i)$

3. If $k=1$ then set $\mathbf{p}^{(1)} = \mathbf{r}^{(1)} = -\mathbf{g}^{(1)}$

4. Compute $\alpha^{(k)}$ using a line search technique [12].

5. Compute $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha^{(k)} \mathbf{p}^{(k)}$ using step 2 to compute

$\mathbf{g}^{(k+1)}$.

6. Compute $\beta^{(k)}$ according to (2.4.8) or (2.4.9) or (2.4.10).
7. Compute $\mathbf{p}^{(k+1)} = -\mathbf{g}^{(k+1)} + \beta^{(k)} \mathbf{p}^{(k)}$.
8. If all the weights are such that the following convergence criterion is satisfied, then go to step 9

$$\sqrt{\frac{\sum_{i=1}^d (E(\mathbf{w}^{(k+1)}))}{d}} < \sqrt{\frac{\sum_{i=1}^d (E(\mathbf{w}^{(k)}))}{d}} < \text{tol}$$

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

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

All other LMS-based training methods can be considered as special cases of $\lambda=0$. For the stopped training method, the stopping criterion is based on the generalization performance of the network, tested using the validation set. The training will be stopped if the generalization error begins to increase. The following off-line training algorithm for the stopped training method is based on Algorithm 2.4.1

Algorithm 3.6.2: Given a set of $S = \{(\mathbf{q}_i, \mathbf{t}_i) | \mathbf{q}_i \text{ is input, } \mathbf{t}_i \text{ is desired output of } \mathbf{q}_i\}$ of d training samples and given a network of K layers with input dimension u and output dimension v .

1. Initialize all weights $\mathbf{w}^{[k]} = (w_{ji}^{[k]})$, $1 = 1, 2, \dots, K$ as random

numbers uniformly distributed between $\frac{-0.5}{\text{fan-in of that unit}}$ and

$\frac{0.5}{\text{fan-in of that unit}}$. Set \mathbf{w}_0, λ .

Initialize $\mathbf{g}^{(k)} = 0$.

- 2. For each sample $(\mathbf{x}_i, t_i) \in S$, repeat the following steps.
 - 2.1 Compute the actual outputs of network according to (3.1.3) and (3.1.4) using the weight $\mathbf{w}^{(k)}$.
 - 2.2 Calculate the gradient $\mathbf{g}(\mathbf{x}_i)$ according to (3.6.3) with

$$p_{ji} = 0.$$

- 2.3 Sum up $\mathbf{g}(\mathbf{x}_i)$, i.e., $\mathbf{g}^{(k)} = \mathbf{g}^{(k)} + \mathbf{g}(\mathbf{x}_i)$

- 3. If $k=1$ then set $\mathbf{p}^{(1)} = \mathbf{r}^{(1)} = -\mathbf{g}^{(1)}$
- 4. Compute $\alpha^{(k)}$ using a line search technique [12].
- 5. Compute $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$ using step 2 to compute $\mathbf{g}^{(k+1)}$.
- 6. Compute $\beta^{(k)}$ according to (2.4.8) or (2.4.9) or (2.4.10)
- 7. Compute $\mathbf{p}^{(k+1)} = -\mathbf{g}^{(k+1)} + \beta^{(k)}\mathbf{p}^{(k)}$
- 8. If $k \bmod C$ (C is constant) $= 0$, calculate the actual output of Networks according to (3.1.3) and (3.1.4) using validation data set and increment v .
- 9. If the following convergence criterion is satisfied using both the validation set data and training set data, then go to step 10

$$\sqrt{\frac{\sum_{i=1}^d (E(\mathbf{w}^{(k+1)}))}{d}} > \sqrt{\frac{\sum_{i=1}^d (E(\mathbf{w}^{(k)}))}{d}}$$

Otherwise set $k=k+1$ and go to step 2

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

The implementation of the stopped training method is problem-dependent. In this thesis, the training will be temporarily stopped after the network has been trained in a constant number of epochs. The network is then to be tested using validation set. If the generalization error decreases, the network resumes the training process, otherwise it stops training.

4. IMPLEMENTATION AND DISCUSSION OF RESULTS

4.1 Language Implementation and Neural Network Architecture Design

In order to test the effectiveness of the penalty method in reducing overfitting in Artificial Neural Networks, we implement it using the A.N.S.I. standard FORTRAN 77 language. The performance of the learning algorithm with penalty method is compared with the performance of the standard learning algorithm without a penalty term.

The design of a neural network is highly problem-dependent. It is the problem that determines what neural network architecture should be used. The topology of the neural network determines the total number of connection weights which in turn determines the performance of the network. Using more connection weights means that we need to have more training samples to train the network in order to get good generalization performance. As a rule of thumb, for a network to be able to generalize, it should have fewer connection weights than there are data points in the training set. Otherwise, overfitting may occur. In this thesis, we first test a small network which doesn't have any overfitting. We then add the hidden nodes to the network. As the network becomes larger, the generalization error becomes larger and larger. By using a penalty method, we can reduce the generalization error.

For a given problem, we need to decide when to stop the training process. There are several stopping criteria. For example, we can use performance function value (RMS) as a criterion. We can set a tolerance value such that the performance function value (RMS) is within the tolerance. We can also use the difference of two consecutive performance function values as the criteria. The problem with these criteria is that we

don't know the generalization performance. A good fitting of the training samples doesn't mean that the network will generalize well over the entire problem domain. Therefore, to obtain better generalization performance, we need to use some optimal stopping point so that the network has good generalization performance. This is especially important when we have a network that has overfitting. In this paper, we will divide the sample data into two sets. One is the training set and the other is the validation set. When the network is trained, we will test the generalization performance at certain numbers of iterations using the validation set. We will use the performance function value as the stopping criterion. If the generalization RMS begins to increase, we will stop training.

4.2 Discussion of Test Results

We use a curve fitting problem to test the learning algorithm. We divide the test data into a training set and a validation set. Each set contains 49 pairs. There are two input node and one output nodes in the network. One hidden layer is used. We increase the number of nodes in the hidden layer from 7 to 20 so that we can test the generalization performance in different network topologies. We are especially interested in testing if the penalty method can improve the generalization performance in an overfitting network.

The initial weights of a neural network have an effect on the training time. Several methods have been proposed to give a neural network as good an initial state as possible. This requires some prior knowledge and/or some understanding of the learning mechanism in the network. We initialize the weights with random values uniformly distributed between -0.5 and 0.5 [15].

Now we analyze the results of the test. First we investigate the network with two input nodes, 7 hidden nodes, and one output node(2/7/1). It has 29 weights. We test the network with different λ values (0.01, 0.001, 0.0001, 0.00001). The training and generalization performance is listed in Table 4.1 through Table 4.5. It takes about 11 epochs of training to get the training RMS value of 0.07078 and generalization RMS value of 0.07247 for λ equals 0. The relationship between generalization RMS and λ is depicted in Figure 4.1. We can see that for λ from 0 to 0.001, there is not much improvement in generalization RMS. The generalization RMS increases with λ larger than 0.001. Next we increase the number of hidden layer nodes to 8 (2/8/1). The network now has 33 weights. The training and generalization RMS are listed in Table 4.6 through Table 4.10. Similarly we test the network with different λ (0.01, 0.001, 0.0001, 0.00001). It takes about 15 epochs to get the training RMS value of 0.07298 and generalization RMS value of 0.0749 for λ equals 0. The relationship between generalization RMS and λ is depicted in Figure 4.2. The generalization RMS is slightly decreased when λ equals 0.001. Next we increase the number of hidden nodes to 9 (2/9/1). This network has 37 weights. The training and generalization RMS are listed in Table 4.11 through Table 4.15. We test the network with different λ (0.01, 0.001, 0.0001, 0.00001). It takes about 12 epochs to get the training RMS value of 0.07598 and generalization RMS value of 0.07966 for λ equals 0. The relationship between generalization RMS and λ is depicted in Figure 4.3. As expected, the generalization RMS is slightly decreased when λ equals 0.0001. These results show that for the network that is not overfitted, if the λ is properly chosen, the generalization of the network can be slightly improved. The maximum improvement is 9% in 2/8/1. The

minimum improvement is 0.5%. The reason for this is that the network is not overfitting yet in these cases. Therefore there is no significant improvement in generalization of the network. The interesting point is that the minimum generalization RMS happens with different λ (for example 2/8/1 and 2/9/1). Another important result is that if λ is not properly chosen, the generalization RMS can increase significantly. The reason is that the penalty term dominates the performance function. In other words the network is over regulated. Now we add another hidden node. The network has 10 hidden nodes (2/10/1). The total number of weights becomes 41, which is very close to the number of sample 49. We test the network with different λ (0.01, 0.008, 0.006, 0.004, 0.002, 0.001, 0.0008, 0.0006, 0.0004, 0.0002, 0.0001, 0.00001). Typical training and generalization RMS are listed in Table 4.16 through Table 4.20. The relationship between generalization RMS and λ is depicted in Figure 4.4. When λ is close to 0.0008, there is a rather large improvement (16.5% in this case) in the generalization performance of the network. Obviously 0.0008 is the optimum point of λ . Also when λ is away from the optimum point, there is no significant change in the generalization behavior of the network. The reason for this phenomenon is that at this point, the network already shows some degree of overfitting. Adding a proper penalty term can indeed improve the generalization performance. Now we further increase the number of hidden nodes to 17 (2/17/1) to force the network to have overfitting. The total number of weights becomes 69, which is larger than the number of data vectors. Again we test the network with different λ (0.01, 0.008, 0.006, 0.004, 0.002, 0.001, 0.0008, 0.0006, 0.0004, 0.0002, 0.0001, 0.00001). Typical training and generalization RMS are listed in Table 4.21 through Table 4.25. The relationship between

generalization RMS and λ is depicted in Figure 4.5. As expected, the generalization performance increases by 21.76% when λ is close to 0.001 which is the optimum point. Again, when λ is away from the optimum point, there is no significant change in the generalization behavior of the network. Finally we increase the number of network hidden nodes to 18 (2/18/1), 19 (2/19/1) and 20 (2/20/1). The training and generalization RMS are listed in Table 4.26 through Table 4.30, Table 4.31 through Table 4.35 and Table 4.36 through Table 4.40 respectively. The relationship between generalization RMS and λ is depicted in Figure 4.6, 4.7 and 4.8 respectively. The generalization performance has increased by 22.66%, 23.01% and 23.58% respectively. The optimum point is 0.001. Again, when λ is away from the optimum point, there is no significant changes in the generalization behavior of the network. The maximum fluctuation is 5%. Figure 4.9 illustrates the relationship between the generalization RMS and number of weights. The generalization RMS increases with the number of weights.

Table 4.1 Performance of Training and Generalization RMS

with 7 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21470	.20951	
1	.81256e-1	.84198e-1	.13344
2	.80169e-1	.83484e-1	.10870e-2
3	.79033e-1	.82563e-1	.11356e-2
4	.77986e-1	.81635e-1	.10473e-2
5	.75036e-1	.78775e-1	.29496e-2
6	.73678e-1	.77430e-1	.13584e-2
7	.72243e-1	.75587e-1	.14346e-2
8	.71906e-1	.73347e-1	.33651e-3
9	.71947e-1	.73375e-1	.41071e-4
10	.70740e-1	.72441e-1	.12078e-2
11	.70781e-1	.72472e-1	.41225e-4

Table 4.2 Performance of Training and Generalization RMS

with 7 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21470	.20951	
1	.81262e-1	.84199e-1	.13344
2	.80162e-1	.83475e-1	.10996e-2
3	.78998e-1	.82525e-1	.11645e-2
4	.77950e-1	.81595e-1	.10472e-2
5	.74875e-1	.78605e-1	.30755e-2
6	.73582e-1	.77323e-1	.12924e-2
7	.72342e-1	.75639e-1	.12400e-2
8	.72368e-1	.73709e-1	.24394e-4
9	.72316e-1	.73740e-1	.47802e-4
10	.70841e-1	.72556e-1	.15252e-2
11	.70896e-1	.72598e-1	.55625e-4

KLAHOMA STATE UNIVERSITY

Table 4.3 Performance of Training and Generalization RMS

with 7 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21470	.20951	
1	.81315e-1	.84208e-1	.13339
2	.80084e-1	.83369e-1	.12310e-2
3	.78629e-1	.82140e-1	.14550e-2
4	.77582e-1	.81189e-1	.10466e-2
5	.73153e-1	.76766e-1	.44289e-2
6	.72514e-1	.76123e-1	.63911e-3
7	.73406e-1	.76018e-1	.89135e-3
8	.74318e-1	.75891e-1	.91268e-3
9	.74394e-1	.75631e-1	.75291e-4
10	.72867e-1	.74796e-1	.15261e-2

Table 4.4 Performance of Training and Generalization RMS

with 7 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.214756	.20951	
1	.818428e-1	.84296e-1	.13291
2	.762907e-1	.75073e-1	.55521e-2
3	.762788e-1	.75064e-1	.11969e-4
4	.728380e-1	.73281e-1	.34407e-2
5	.740105e-1	.73026e-1	.11724e-2
6	.729603e-1	.72207e-1	.10502e-2
7	.726419e-1	.71966e-1	.31834e-3
8	.728269e-1	.72103e-1	.18496e-3

Table 4.5 Performance of Training and Generalization RMS

with 7 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21523	.20951	
1	.86686e-1	.85248e-1	.12854
2	.86658e-1	.85241e-1	.27226e-4
3	.86659e-1	.85241e-1	.54052e-7

OKLAHOMA STATE UNIVERSITY

Table 4.6 Performance of Training and Generalization RMS

with 8 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21300	.20783	
1	.81865e-1	.84872e-1	.13113
2	.81546e-1	.84750e-1	.31916e-3
3	.81314e-1	.84631e-1	.23208e-3
4	.80831e-1	.84302e-1	.48292e-3
5	.79667e-1	.83330e-1	.11635e-2
6	.77924e-1	.81717e-1	.17428e-2
7	.73846e-1	.77607e-1	.40786e-2
8	.73020e-1	.76771e-1	.82560e-3
9	.73335e-1	.77067e-1	.31477e-3
10	.73138e-1	.76587e-1	.19700e-3
11	.73397e-1	.76834e-1	.25948e-3
12	.73596e-1	.75347e-1	.19911e-3
13	.73689e-1	.75399e-1	.92867e-4
14	.72979e-1	.74904e-1	.71044e-3
15	.72989e-1	.74911e-1	.10639e-4

Table 4.7 Performance of Training and Generalization RMS

with 8 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21300	.20783	
1	.81870e-1	.84873e-1	.13113
2	.81550e-1	.84749e-1	.32092e-3
3	.81312e-1	.84626e-1	.23800e-3
4	.80803e-1	.84276e-1	.50863e-3
5	.79611e-1	.83275e-1	.11915e-2
6	.77787e-1	.81577e-1	.18243e-2
7	.73775e-1	.77526e-1	.40117e-2
8	.72945e-1	.76684e-1	.83007e-3
9	.73206e-1	.76931e-1	.26036e-3
10	.73080e-1	.76556e-1	.12506e-3
11	.73488e-1	.76945e-1	.40746e-3

OKLAHOMA STATE UNIVERSITY

Table 4.8 Performance of Training and Generalization RMS

with 8 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21300	.20783	
1	.81922e-1	.84879e-1	.13108
2	.81583e-1	.84743e-1	.33856e-3
3	.81278e-1	.84572e-1	.30554e-3
4	.80455e-1	.83948e-1	.82285e-3
5	.79244e-1	.82891e-1	.12103e-2
6	.75561e-1	.79263e-1	.36829e-2
7	.73676e-1	.77358e-1	.18849e-2
8	.72589e-1	.76121e-1	.10873e-2
9	.72426e-1	.75774e-1	.16327e-2
10	.72028e-1	.75375e-1	.39757e-3
11	.73457e-1	.76553e-1	.14292e-2

Table 4.9 Performance of Training and Generalization RMS

with 8 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalization RMS	Convergence Error
0	.21306	.20783	
1	.82428e-1	.84938e-1	.13063
2	.73557e-1	.71637e-1	.88708e-2
3	.73553e-1	.71632e-1	.48850e-5
4	.69478e-1	.67971e-1	.40742e-2
5	.69746e-1	.68204e-1	.26743e-3
6	.69678e-1	.61845e-1	.67486e-4
7	.69682e-1	.68148e-1	.37051e-5
8	.69682e-1	.68148e-1	.30886e-6
9	.69682e-1	.68148e-1	.23042e-7

Table 4.10 Performance of Training and Generalization RMS

with 8 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21360	.20783	
1	.87072e-1	.85567e-1	.12653
2	.87065e-1	.85565e-1	.73488e-5
3	.87065e-1	.85565e-1	.46825e-7

OKLAHOMA STATE UNIVERSITY

Table 4.11 Performance of Training and Generalization RMS

with 9 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21164	.20649	
1	.82246e-1	.85295e-1	.12939
2	.82098e-1	.85269e-1	.14840e-3
3	.82037e-1	.85255e-1	.61313e-4
4	.81982e-1	.85238e-1	.54493e-4
5	.81922e-1	.85216e-1	.60419e-4
6	.81830e-1	.85176e-1	.91445e-4
7	.81616e-1	.85053e-1	.21424e-3
8	.80688e-1	.84328e-1	.92840e-3
9	.79370e-1	.83157e-1	.13175e-2
10	.75564e-1	.79274e-1	.38059e-2
11	.75491e-1	.79203e-1	.73353e-4
12	.75979e-1	.79660e-1	.48856e-3

Table 4.12 Performance of Training and Generalization RMS

with 9 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21164	.20649	
1	.82252e-1	.85295e-1	.12939
2	.82103e-1	.85269e-1	.14878e-3
3	.82041e-1	.85255e-1	.62247e-4
4	.81985e-1	.85238e-1	.56319e-4
5	.81920e-1	.85214e-1	.64392e-4
6	.81818e-1	.85166e-1	.10264e-3
7	.81553e-1	.85007e-1	.26484e-3
8	.80362e-1	.84034e-1	.11909e-2
9	.79113e-1	.82908e-1	.12490e-2
10	.75376e-1	.78869e-1	.37366e-2
11	.75933e-1	.78988e-1	.55684e-3
12	.76266e-1	.78714e-1	.33293e-3
13	.76260e-1	.78711e-1	.61392e-5
14	.76303e-1	.78735e-1	.43153e-4

OKLAHOMA STATE UNIVERSITY

Table 4.13 Performance of Training and Generalization RMS

with 9 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21165	.20649	
1	.82301e-1	.85300e-1	.12934
2	.82149e-1	.85271e-1	.15257e-3
3	.82077e-1	.85253e-1	.72151e-4
4	.81998e-1	.85225e-1	.78343e-4
5	.81874e-1	.85168e-1	.12462e-3
6	.81506e-1	.84933e-1	.36810e-3
7	.80028e-1	.83674e-1	.14773e-2
8	.78732e-1	.82474e-1	.12961e-2
9	.75209e-1	.78455e-1	.35231e-2
10	.75985e-1	.78188e-1	.77556e-3
11	.76001e-1	.78197e-1	.16124e-4

Table 4.14 Performance of Training and Generalization RMS

with 9 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21171	.20649	
1	.82792e-1	.85343e-1	.12891
2	.82426e-1	.85191e-1	.36665e-3
3	.80899e-1	.81993e-1	.15264e-2
4	.80905e-1	.81983e-1	.62907e-2
5	.78635e-1	.80548e-1	.22705e-2
6	.78620e-1	.80540e-1	.15373e-4
7	.78616e-1	.80539e-1	.31036e-5
8	.78616e-1	.80539e-1	.17509e-6
9	.78616e-1	.80539e-1	.56642e-8
10	.78616e-1	.80539e-1	.19099e-8
11	.78616e-1	.80539e-1	.23701e-8
12	.78616e-1	.80539e-1	.23900e-8
13	.78616e-1	.80539e-1	.24640e-8

OKLAHOMA STATE UNIVERSITY

Table 4.15 Performance of Training and Generalization RMS

with 9 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	convergence error
0	.21231	.20649	
1	.87290e-1	.85800e-1	.12502
2	.87298e-1	.85803e-1	.80299e-5
3	.87298e-1	.85803e-1	.29196e-7
4	.87298e-1	.85803e-1	.72447e-9
5	.87298e-1	.85803e-1	.10729e-9
6	.87298e-1	.85803e-1	.11008e-8
7	.87298e-1	.85803e-1	.11066e-8

Table 4.16 Performance of Training and Generalization RMS

with 10 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21053	.20540	
1	.82509e-1	.85585e-1	.12802
2	.82425e-1	.85584e-1	.84206e-4
3	.82402e-1	.85583e-1	.22950e-4
4	.82391e-1	.85583e-1	.11260e-4
5	.82338e-1	.85583e-1	.54434e-5
6	.82383e-1	.85583e-1	.26885e-5
7	.82381e-1	.85583e-1	.13347e-5
8	.82381e-1	.85583e-1	.66744e-6
9	.82380e-1	.85583e-1	.33381e-6
10	.82380e-1	.85583e-1	.16989e-6
11	.82380e-1	.85583e-1	.83724e-7
12	.82380e-1	.85583e-1	.43954e-7

OKLAHOMA STATE UNIVERSITY

Table 4.17 Performance of Training and Generalization RMS

with 10 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21053	.20540	
1	.82514e-1	.85585e-1	.12802
2	.82430e-1	.85584e-1	.84214e-4
3	.82407e-1	.85584e-1	.23136e-4
4	.82396e-1	.85583e-1	.11472e-4
5	.82390e-1	.85583e-1	.56202e-5
6	.82387e-1	.85583e-1	.28169e-5
7	.82386e-1	.85583e-1	.14299e-5
8	.82385e-1	.85583e-1	.72607e-6
9	.82385e-1	.85583e-1	.37244e-6
10	.82384e-1	.85583e-1	.18811e-6
11	.82384e-1	.85583e-1	.95430e-7
12	.82384e-1	.85583e-1	.50804e-7
13	.82384e-1	.85583e-1	.23701e-7

Table 4.18 Performance of Training and Generalization RMS

with 10 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21054	.20540	
1	.82563e-1	.85589e-1	.12798
2	.82478e-1	.85587e-1	.84371e-4
3	.82453e-1	.85586e-1	.25065e-4
4	.82439e-1	.85586e-1	.13688e-4
5	.82432e-1	.85585e-1	.76576e-5
6	.82427e-1	.85585e-1	.44965e-5
7	.82425e-1	.85585e-1	.27137e-5
8	.82423e-1	.85584e-1	.16627e-5
9	.82422e-1	.85584e-1	.10320e-5
10	.82421e-1	.85584e-1	.64469e-6
11	.82421e-1	.85584e-1	.40640e-6
12	.82421e-1	.85584e-1	.25965e-6
13	.82420e-1	.85584e-1	.15910e-6
14	.82420e-1	.85584e-1	.10113e-6
15	.82420e-1	.85584e-1	.65429e-7
16	.82420e-1	.85584e-1	.37626e-7
17	.82420e-1	.85584e-1	.26195e-7
18	.82420e-1	.85584e-1	.15057e-7
19	.82420e-1	.85584e-1	.13412e-7

Table 4.19 Performance of Training and Generalization RMS

with 10 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21061	.20540	
1	.83040e-1	.85622e-1	.12757
2	.82927e-1	.85612e-1	.11260e-3
3	.82304e-1	.85251e-1	.62384e-3
4	.75336e-1	.74562e-1	.69670e-2
5	.75338e-1	.74563e-1	.18460e-5

Table 4.20 Performance of Training and Generalization RMS

with 10 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.21128	.20540	
1	.87416e-1	.85975e-1	.12386
2	.87439e-1	.85982e-1	.22154e-4

Table 4.21 Performance of Training and Generalization RMS

with 17 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergent error
0	.20624	.20116	
1	.83251e-1	.86404e-1	.12299
2	.83242e-1	.86409e-1	.93941e-5
3	.83241e-1	.86409e-1	.50516e-6
4	.83241e-1	.86409e-1	.32754e-7
5	.83241e-1	.86409e-1	.95812e-9

Table 4.22 Performance of Training and Generalization RMS

with 17 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20624	.20116	
1	.83256e-1	.86404e-1	.12298
2	.83247e-1	.86409e-1	.92395e-5
3	.83246e-1	.86409e-1	.49143e-6
4	.83246e-1	.86409e-1	.32637e-7
5	.83246e-1	.86409e-1	.22227e-9
6	.83246e-1	.86409e-1	.15910e-8

Table 4.23 Performance of Training and Generalization RMS

with 17 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20625	.20116	
1	.83302e-1	.86405e-1	.12295
2	.83294e-1	.86409e-1	.78463e-5
3	.83294e-1	.86410e-1	.37290e-6
4	.83294e-1	.86410e-1	.23036e-7
5	.83294e-1	.86410e-1	.10274e-8
6	.83294e-1	.86410e-1	.37958e-10

Table 4.24 Performance of Training and Generalization RMS

with 17 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalization RMS	Convergent error
0	.20636	.20116	
1	.15497	.16212	.51394e-1
2	.15494	.16212	.33582e-4
3	.15494	.16212	.52376e-8
4	.15494	.16212	.52370e-8
5	.15494	.16212	.87815e-9
6	.15494	.16212	.90276e-7
7	.88951e-1	.91593e-1	.65989e-1
8	.88948e-1	.91568e-1	.34102e-5
9	.85611e-1	.83072e-1	.33362e-2
10	.82904e-1	.80588e-1	.27070e-2
11	.68028e-1	.66880e-1	.14875e-1
12	.68843e-1	.67574e-1	.81435e-3

Table 4.25 Performance of Training and Generalization RMS

with 17 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20751	.20116	
1	.16007	.16212	.47439e-1
2	.16003	.16212	.47439e-1
3	.16006	.16212	.20426e-6
4	.92736e-1	.88925e-1	.67297e-1
5	.92736e-1	.88926e-1	.15556e-6
6	.92737e-1	.88911e-1	.15256e-5
7	.92737e-1	.88912e-1	.27432e-7

Table 4.26 Performance of Training and Generalization RMS

with 18 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20588	.20082	
1	.83299e-1	.86456e-1	.12258
2	.83291e-1	.86460e-1	.78597e-5
3	.83290e-1	.86461e-1	.35343e-6
4	.83290e-1	.86461e-1	.20773e-7
5	.83290e-1	.86461e-1	.10029e-8

Table 4.27 Performance of Training and Generalization RMS

with 18 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20588	.20082	
1	.83304e-1	.86456e-1	.12258
2	.83296e-1	.86460e-1	.77094e-5
3	.83296e-1	.86461e-1	.34352e-6
4	.83296e-1	.86461e-1	.22254e-7
5	.83296e-1	.86461e-1	.33955e-8

OKLAHOMA STATE UNIVERSITY

Table 4.28 Performance of Training and Generalization RMS

with 18 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20590	.20082	
1	.83350e-1	.86457e-1	.12255
2	.83343e-1	.86461e-1	.63923e-5
3	.83343e-1	.86461e-1	.24713e-6
4	.83343e-1	.86461e-1	.13728e-7
5	.83343e-1	.86461e-1	.18923e-8

Table 4.29 Performance of Training and Generalization RMS

with 18 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20602	.20082	
1	.15500	.16212	.51014e-1
2	.15497	.16212	.34231e-4
3	.15497	.16212	.40106e-8
4	.15497	.16212	.40101e-8
5	.15497	.16212	.85814e-10
6	.15497	.16212	.10421e-6
7	.89342e-1	.91949e-1	.65631e-1
8	.89342e-1	.91954e-1	.71895e-6
9	.86015e-1	.83390e-1	.33271e-2
10	.83447e-1	.81007e-1	.25680e-2
11	.69512e-1	.65347e-1	.13935e-1
12	.71419e-1	.67266e-1	.19075e-2

Table 4.30 Performance of Training and Generalization RMS

with 18 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20723	.20082	
1	.16038	.16212	.46849e-1
2	.16034	.16212	.38959e-4
3	.16034	.16212	.21797e-6
4	.92795e-1	.88842e-1	.67549e-1
5	.92796e-1	.88841e-1	.19225e-6
6	.92797e-1	.88831e-1	.13584e-5
7	.92797e-1	.88831e-1	.21081e-7
8	.92797e-1	.88831e-1	.12166e-8

Table 4.31 Performance of Training and Generalization RMS

with 19 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20556	.20050	
1	.83340e-1	.86501e-1	.12222
2	.83333e-1	.86505e-1	.64557e-5
3	.83333e-1	.86505e-1	.24511e-6
4	.83333e-1	.86505e-1	.11280e-7
5	.83333e-1	.86505e-1	.28768e-9

Table 4.32 Performance of Training and Generalization RMS

with 19 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20557	.20050	
1	.83345e-1	.86501e-1	.12222
2	.83339e-1	.86505e-1	.63169e-5
3	.83338e-1	.86505e-1	.23887e-6
4	.83338e-1	.86505e-1	.14396e-7
5	.83338e-1	.86505e-1	.34601e-8

Table 4.33 Performance of Training and Generalization RMS

with 19 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20558	.20050	
1	.83392e-1	.86502e-1	.12219
2	.83387e-1	.86505e-1	.50733e-5
3	.83386e-1	.86505e-1	.16458e-6
4	.83386e-1	.86505e-1	.90807e-8
5	.83386e-1	.86505e-1	.28287e-8

Table 4.34 Performance of Training and Generalization RMS

with 19 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20571	.20050	
1	.15504	.16212	.50670e-1
2	.15500	.16212	.34791e-4
3	.15500	.16212	.69051e-8
4	.15500	.16212	.79955e-8
5	.15500	.16212	.40426e-8
6	.89710e-1	.92336e-1	.65296e-1
7	.89703e-1	.92302e-1	.66916e-5
8	.85541e-1	.82931e-1	.41623e-2
9	.83212e-1	.80810e-1	.23292e-2
10	.69514e-1	.65436e-1	.13697e-1
11	.70726e-1	.66649e-1	.12120e-2

Table 4.35 Performance of Training and Generalization RMS

with 19 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20699	.20050	
1	.16069	.16212	.46295e-1
2	.16065	.16212	.38828e-4
3	.16065	.16212	.22826e-6
4	.92840e-1	.88759e-1	.67815e-1
5	.92840e-1	.88762e-1	.65271e-6
6	.92841e-1	.88756e-1	.10791e-5
7	.92841e-1	.88756e-1	.12603e-7
8	.92841e-1	.88756e-1	.32911e-8

Table 4.36 Performance of Training and Generalization RMS

with 20 hidden nodes and $\lambda = 0$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20528	.20022	
1	.83376e-1	.86541e-1	.12190
2	.83370e-1	.86544e-1	.53513e-5
3	.83370e-1	.86545e-1	.17393e-6
4	.83370e-1	.86545e-1	.41071e-8
5	.83370e-1	.86545e-1	.28875e-8

Table 4.37 Performance of Training and Generalization RMS

with 20 hidden nodes and $\lambda = 0.00001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20528	.20022	
1	.83381e-1	.86541e-1	.12190
2	.83376e-1	.86544e-1	.52143e-5
3	.83376e-1	.86545e-1	.16957e-6
4	.83376e-1	.86545e-1	.74668e-8
5	.83376e-1	.86545e-1	.92002e-9

Table 4.38 Performance of Training and Generalization RMS

with 20 hidden nodes and $\lambda = 0.0001$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20529	.20022	
1	.83428e-1	.86542e-1	.12186
2	.83424e-1	.86545e-1	.40386e-5
3	.83424e-1	.86545e-1	.10815e-6
4	.83424e-1	.86545e-1	.64594e-9

Table 4.39 Performance of Training and Generalization RMS

with 20 hidden nodes and $\lambda = 0.001$

Epoch	Training RMS	Generalizatin RMS	Convergence error
0	.20543	.20022	
1	.15507	.16212	.50356e-1
2	.15503	.16212	.35301e-4
3	.15503	.16212	.26528e-8
4	.15503	.16212	.26525e-8
5	.45503e-1	.16212	.12365e-8
6	.90054e-1	.92656e-1	.64985e-1
7	90049e-1	.92634e-1	.47295e-5
8	.85303e-1	.82694e-1	.47460e-2
9	83209e-1	.80812e-1	.20942e-2
10	.69532e-1	.65542e-1	.13677e-1
11	.69788e-1	.66073e-1	.25598e-3

Table 4.40 Performance of Training and Generalization RMS

with 20 hidden nodes and $\lambda = 0.01$

Epoch	Training RMS	Generalization RMS	Convergence error
0	.20677	.20022	
1	.16100	.16212	.45772e-1
2	.16096	.16212	.38782e-4
3	.16096	.16212	.24146e-6
4	.92872e-1	.88685e-1	.68094e-1
5	.92870e-1	.88692e-1	.15384e-5
6	.92871e-1	.88688e-1	.74493e-6
7	.92871e-1	.88688e-1	.10809e-7
8	.92871e-1	.88688e-1	.37280e-9

UNIVERSITY OF CALIFORNIA, BERKELEY

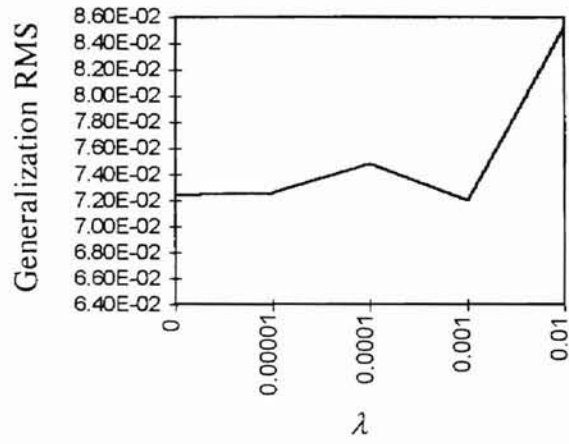


Figure 4.1 The Relationship Between Generalization RMS and λ (2/7/1).

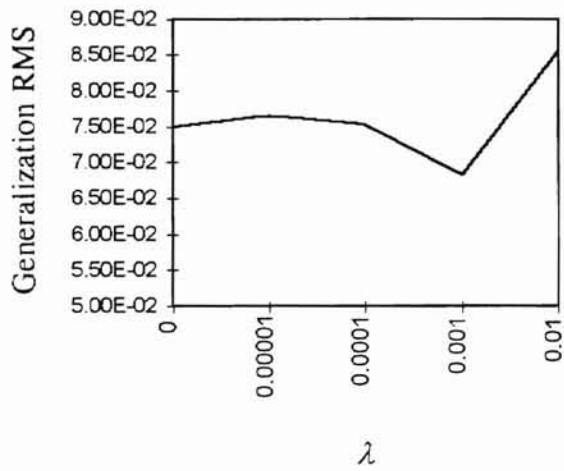


Figure 4.2 The Relationship Between Generalization RMS and λ (2/8/1).

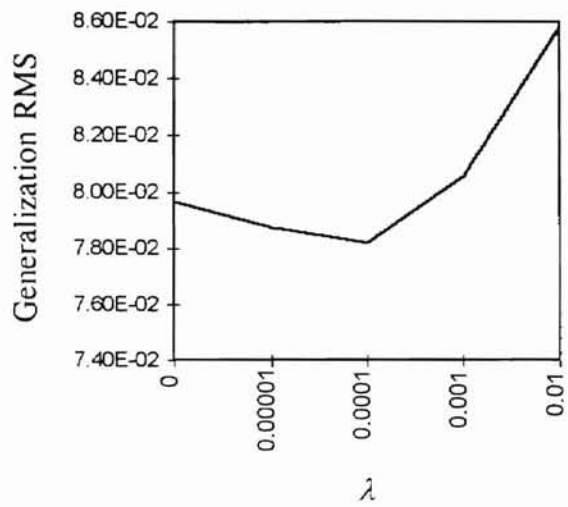


Figure 4.3 The Relationship Between Generalization RMS and λ (2/9/1).

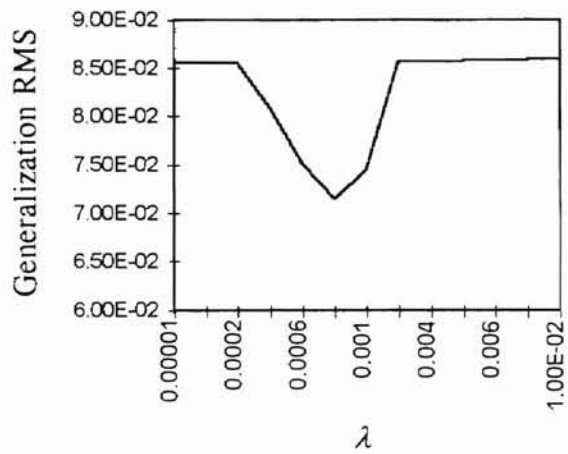


Figure 4.4 The Relationship Between Generalization RMS and λ (2/10/1).

UNIVERSITÄT DUISBURG ESSEN

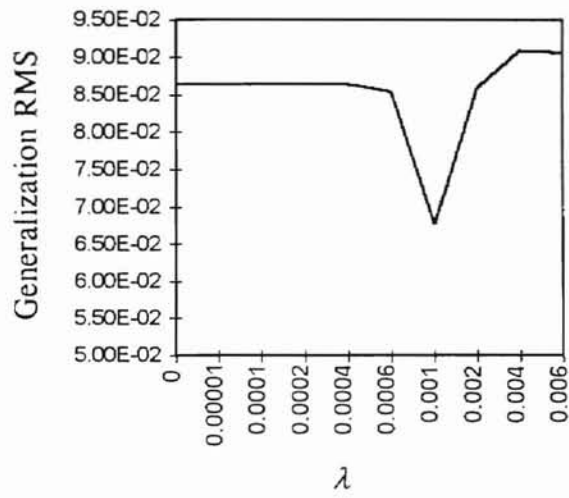


Figure 4.5 The Relationship Between Generalization RMS and λ (2/17/1).

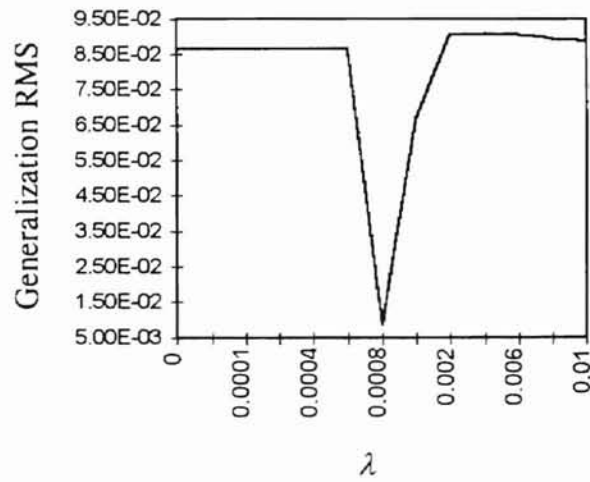


Figure 4.6 The relationship Between Generalization RMS and λ (2/18/1).

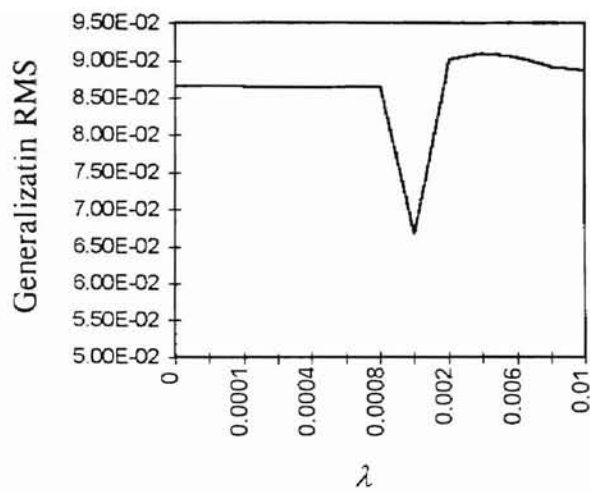


Figure 4.7 The Relationship Between Generalization RMS and λ (2/19/1).

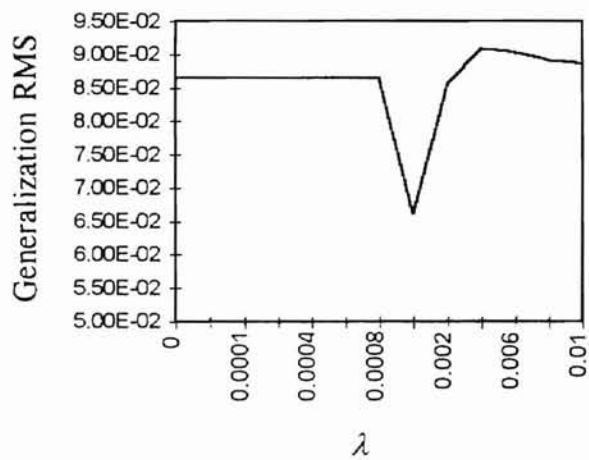


Figure 4.8 The relationship Between Generalization RMS and λ (2/20/1).

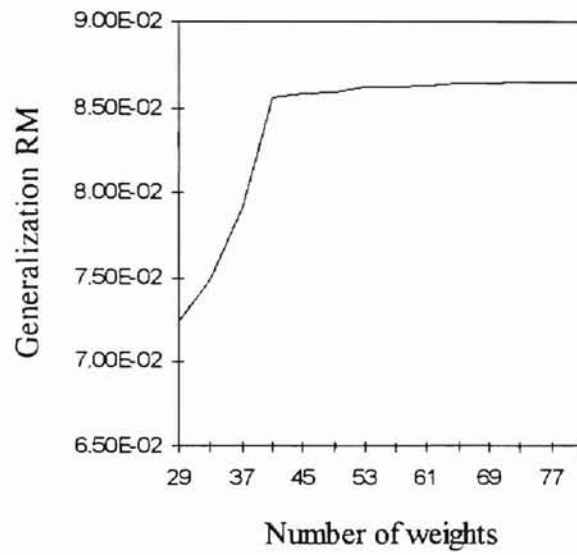


Figure 4.9 The Relationship Between Generalization RMS and Number of Weights

5. CONCLUSION AND FUTURE WORK

Overfitting is a very important issue in artificial neural networks. A network that cannot generalize is useless. There are several methods to reduce overfitting. In this paper, we use a penalty method to reduce the overfitting. The results are compared with those without penalty term. From this study, we find several important conclusions

- Overfitting does exist in artificial neural networks.
- As the neural network becomes larger, the generalization performance becomes worse. So we may choose the smallest networks that fit the data.
- When the network has a larger number of samples than weights, the penalty method can still be used to increase slightly the generalization performance of the network. However, we should be careful in choosing λ to be close to the optimum point. Otherwise, generalization performance can be decreased significantly.
- When the network has more weights than number of samples, the penalty method can be used to improve significantly the generalization performance of the networks. We need to choose λ close to the optimum point to improve the generalization performance. However, generally speaking, the performance will not be significantly changed if λ is not close to the optimum point.
- The optimum point of λ is network architecture dependent.

Future work can be done in several areas as listed below:

- To use different penalty terms. One example is to include the output term in the performance function. Another example is to include both output term and the

weight term [24] or to use a roughness penalty [20].

- Another method that can be investigated is an interactive method in which the designer checks the trained network and decides which nodes to remove.

Several heuristics are used to identify units that don't contribute to the solution.

One method is to remove a node that has a constant output over all training patterns. When a number of nodes have highly correlated responses over all patterns, they can be combined into one node.

- A comparison study may be needed to investigate the effectiveness of different methods in reducing overfitting.

Bibliography

- [1] John, H., K. Anders and G. P. Richard "Introduction to the Theory of Neural Computers", Lecture Notes Vol. I. Addison-Wesley Publishing Company.. 1991.
- [2] Hecht-Nielsen Robert, "Neurocomputing", Addison-Wesley Publishing Company. 1990.
- [3] McCulloch, W. S. and W. Pitts., "A Logical Calculus of the Ideas Immanent in Nervous Activity", Bulletin of Math. Bio., 5, 1943.
- [4] Hebb, D., "The Organization of Behavior," Wiley, New York, 1949.
- [5] Minsky, M., "Neural Nets and the Brain-model Problem", Doctoral Dissertation. Princeton University, Princeton, NJ, 1954.
- [6] Rosenblatt, F., "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain", Psych. Rev., 65, 1958.
- [7] Minsky, M. and S. Papert, "Perceptrons", MIT Press. Cambridge, MA. 1969.
- [8] Hopfield, J.J., "Neurons with Graded-response Have Collective Computational Properties Like Those Two-state Neurons", Proc. Natl. Acad. Sci. 81, 1984.
- [9] Hopfield, J.J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", Proc. Natl. Acad. Sci. 79, 1982.
- [10] Rumelhart, D.E. and J. L. McClelland, "Parallel Distributed Processing: Explorations in the Micro Structure of Cognition I & II," MIT Press. Cambridge MA. 1986.
- [11] Strang G., "Linear Algebra and its Application", Academic Press, New York. 1980.
- [12] Scales, L. E., "Introduction to Nonlinear Optimization", New York, Springer-Verlag.

1985.

- [13] Magnus, R. H., "Conjugate Direction Methods in Optimization". Springer-Verlag, New York, 1980.
- [14] Wolfe, M.A., "Numerical Methods for Unconstrained Optimization". Van Nostrand Reinhold Company, 1978.
- [15] Cichocki, A. and Unbehauen, R., "Neural Networks for Optimization and Signal Processing", Wiley, 1993.
- [16] Hagan Martin T., "Neural Network Design", Lecture Notes, Oklahoma State University, 1995.
- [17] Freeman James A. and David M. Skapura, "Neural Networks Algorithms, Applications and Programming Techniques", Addison-Wesley Publishing Company, 1992.
- [18] Barnard Etienne, "Optimization for Training Neural Nets", IEEE Transactions on Neural Networks, Vol. 3, No. 2, pp. 232-240., Mar., 1992.
- [19] Webb Andrew R., "Functional Approximation by Feedforward Networks: A Least-squares Approach to Neural Networks", IEEE Transactions on Neural Networks, Vol. 5, No. 3, pp. 363-371, May, 1994.
- [20] Bishop Chris M., "Curvature-driven Smoothing: A Learning Algorithm for Feedforward Networks", IEEE Transactions on Neural Networks, Vol. 4, No. 5, pp. 882-884, Sept. 1993.
- [21] De Villiers Jacques and Etienne Barnard " Backpropagation Neural Nets with One and Two Hidden Layers", IEEE Transactions on Neural Networks, Vol. 4, No. 1, pp.

136 - 141, Jan. 1992.

- [22] Hagan Martin T., "Training Feedforward Networks with the Marquardt Algorithm", IEEE Transactions on Neural Networks, Vol. 5, No. 6, pp. 989-993, Nov. 1994.
- [23] Reed Russell, "Pruning Algorithms-A Survey", IEEE Transactions on Neural Networks, Vol. 4, No. 5, 1993.
- [24] Weigend Andreas S., Bernardo A. Huberman and David E. Rumelhart, "Generalization by Weight-Elimination Applied to Currency Exchange Rate Prediction", Proc. Int. Joint Conf. Neural Networks, Vol. I, pp. 837-841, Seattle, 1991.
- [25] Amirikian Bagrat and Hajime Nishimura, "What Size Network Is Good for Generalization of a Specific Task of Interest?", Neural Network, Vol. 7, No. 2, pp. 321-329, 1994.
- [26] Chauvin Yves, "Generalization Performance of Overtrained Back-propagation Networks", in Lecture Notes in Computer Science, Edited by L. B. Almeida and C. J. Wellekens, Springer-Verlag, 1990.
- [27] Press William H., Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, "Numerical Recipes in FORTRAN", Cambridge University Press, 1992.
- [28] Levenberg Kenneth., "A Method For the Solution of Certain Non-linear Problems in Least Squares", Quart. Appl. Math., No. 2, pp. 164 - 168, 1944.
- [29] Marquardt Donald W., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters", J. Soc. Indust. Appl. Math. Vol. 11, No. 2, pp. 431 - 441, June, 1963.

- [30] Ackley David H. and Michael L. Littman, "Generalization and Scaling in Reinforcement Learning", in *Advances in Neural Information Processing 2*, D.S. Touretzky, Ed. pp. 550 - 557, 1989.
- [31] Mozer Michael C. and Paul Smolensky, "Skeletonization: A Technique for Trimming the Fat From a Network via Relevance Assessment", in *Advances in Neural Information Processing 1*, D.S. Touretzky, Ed. pp. 107 - 115, 1989.
- [33] Baum Eric B. and David Haussler, "What Size Net Gives Valid Generalization?", in *Advances in Neural Information Processing 1*, D.S. Touretzky, Ed. pp. 81 - 90, 1989.
- [34] Chauvin Yves., "A Back-Propagation Algorithm With Optimal Use of Hidden Units", in *Advances in Neural Information Processing 1*, D.S. Touretzky, Ed. pp. 519 - 526, 1989.
- [35] Ahmad Subatai and Gerald Tesauro, "Scaling and Generalization in Neural Networks: A Case Study", in *Advances in Neural Information Processing 1*, D.S. Touretzky, Ed. pp. 160 - 168, 1989.
- [36] Morgan, N. and H. Bourlard, "Generalization and Parameter Estimation in Feedforward Nets: Some Experiments", in *Advances in Neural Information Processing 2*, D.S. Touretzky, Ed. pp. 630 - 637, 1989.
- [37] Hanson Stephen Jose and Lorien Y. Pratt, "Comparing Biases Minimal Network Construction with Back-Propagation", in *Advances in Neural Information Processing 1*, D.S. Touretzky, Ed. pp. 177 - 185, 1989.
- [38] Chauvin Yves, "Dynamic Behavior of Constrained Back-Propagation Networks", in

- Advances in Neural Information Processing 2, D.S. Touretzky, Ed. pp. 642 - 649, 1989.
- [39] Le Cun Yann, John S. Denker and Sara A. Solla, "Optimal Brain Damage", in Advances in Neural Information Processing 2, D.S. Touretzky, Ed. pp. 598 - 605, 1989.
- [40] Dietterich Tom, "Overfitting and Undercomputing in Machine Learning", ACM Computing Survey, Vol. 27, No. 3, pp. 326 - 327, Sept. 1995.
- [41] Marquardt Donald W., "Generalized Inverse, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation", Technometrics, Vol. 12, No. 3, pp. 591 - 612, August, 1970.
- [42] Marquardt Donald W. and Donald D. Snee, "Ridge Regression in Practice", The American Statistician, Vol. 29, No. 1, pp. 3 - 19, Feb. 1975.
- [43] Hoerl Arthur E. and Robert W. Kennard, "Ridge Regression: Biased Estimation for Nonorthogonal Problems", Technometrics, Vol. 12, No. 1, pp. 55 - 67., Feb. 1970.
- [44] Hoerl Arthur E. and Robert W. Kennard, "Ridge Regression: Applications to Nonorthogonal Problems", Technometrics, Vol. 12, No. 1, pp. 69 - 82, Feb. 1970.
- [45] Sietsman, J. and R.J.F Dow, "Neural Net Pruning - Why and How", IEEE International Conference on Neural Network 1, San Diego, California, pp.325 - 333, July, 1988.
- [46] Green, P.J. and B. W. Silberman, "Nonparametric Regression and Generalized Linear Models - A Roughness Penalty Approach", Chapman & Hall. 1994.

APPENDIX--PROGRAM LISTING

```

PROGRAM DRIVER
C*****
C   THIS DRIVER IS TO GENERATE THE RANDOM WEIGHTS      *
C   W(MLAYR, MNODE, 0:MNODE) --THE WEIGHT OF          *
C   EACH LAYER.                                       *
C   P(MNODE) -- THE INPUT DATA OF THE SAMPLE        *
C   O(MNODE) -- THE OUTPUT CALCULATED FROM THE INPUT *
C   DATA SAMPLE.                                     *
C   N(MLAYR, MNODE) -- THE WEIGHTED SUM OF THE       *
C   INPUTS OF A NEURON MNODE IN LAYER MLAYR         *
C   REF (3.1.1)                                       *
C   A(0:MLAYR, 0:MNODE) -- THE OUTPUT OF THE NEURON *
C   MNODE IN LAYER MLAYR. REF (3.1.2)               *
C   NOTICE THAT A(0,*) REPRESENTS THE INPUT        *
C   LAYER. A(*,0) REPRESENTS THE BIAS.              *
C   NNODE(0:MLAYR) -- THE NUMBER OF NODE IN EACH   *
C   LAYER.                                           *
C   LAYER -- THE ACTUAL TOTAL LAYER OF THE NET. (EXCLUDING *
C   THE INPUT LAYER)                                *
C   MLAYR -- THE MAXMUM LAYER A NET CAN HAVE.      *
C   MNODE -- THE MAXMUM NODE ONE LAYER OF A NET CAN HAVE *
C   LL -- SAMPLE INDEX                               *
C*****
PARAMETER(MLAYR = 4, MNODE = 100,MSAMP = 200)
DOUBLE PRECISION DRANDOM, W(MLAYR, MNODE, 0:MNODE),
+ SEED,TOL,W0,LAMDA,P(MSAMP,MNODE),O(MSAMP,MNODE),
+ N(MLAYR,MNODE), A(0:MLAYR,0:MNODE),
+ SENSI(MLAYR,MNODE),T(MSAMP,MNODE),ERROR2,ERROR1,
+ G(MLAYR,MNODE,0:MNODE),TG(MLAYR,MNODE,0:MNODE),
+ FRET,TOL1,ERROR
INTEGER K,I,J,LL,NNODE(0:MLAYR),METHOD,LAYER,NSAMP
INTEGER NUM,ITER,MAXNUM,NWEIG,PSTAT
C
C   THE FOLLOWING DATA IS USED IN CONJUGATE GRADIENT METHOD
C
DOUBLE PRECISION PP(MLAYR,MNODE,0:MNODE),BETA,
+ TG0(MLAYR,MNODE,0:MNODE),PP0(MLAYR,MNODE,0:MNODE)
PSTAT=10
C
C   SET UP NETWORK
C
CALL NETSETUP(LAYER,MLAYR,NNODE,SEED,W0,LAMDA,METHOD)
CALL NETPRINT(LAYER,MLAYR,NNODE,SEED,W0,LAMDA,METHOD)
C
C   INITIAL WEIGHT WITH RANDOM NUMBER.
C
CALL INIWEIGHT(W, LAYER, MLAYR.NNODE, MNODE,SEED,
+ NWEIG)
C
C   PRINT THE NUMBER OF WEIGHT
C
WRITE(*,1001)NWEIG
1001 FORMAT(1X,'THE NUMBER OF WEIGHT IS. ',I5)
C

```

```

C   READ IN TRAINING DATA P(I) AND T(I)
C   READ IN THE INPUT AND DESIRED OUTPUT OF ONE TRAINING SAMPLE
C
C   CALL GETINPUTDATA(P,T,MNODE,NNODE(0),NNODE(LAYER),
+ MSAMP,NSAMP)
C
C   CALCULATE THE PERFORMANCE FUNCTION
C
C   ERROR = SQRT(TEST(MSAMP,MNODE,W,MLAYR,LAYER,
+ NNODE,LAMDA,W0)/NSAMP)
C   PRINT*, 'BEFORE TRAINING GENERALIZATION ERROR: ',ERROR
C
C   LOOP OVER ITERATION
C   SET TOLERANCE AND MAXIMUM ITERATION NUMBER
C
C   TOL = 4.0D-10
C   TOL1 = 3.5D-2
C   MAXITER=20
C   ITER=0
C
C   1000  ITER = ITER + 1
C
C   ENTER ITERATION
C
C   CALL INITG(LAYER,MLAYR,NNODE,MNODE,TG)
C
C   SUM TOTAL GRADIENT
C
C   DO 320 LL=1,NSAMP
C
C   FEEDFORWARD COMPUTATION
C
C   CALL FORWARD(P,O,N,MLAYR,LAYER,MNODE,A,
+ NNODE,W,LL,MSAMP)
C
C   CALCULATE THE SENSITIVITY MATRIX
C
C   CALL SENSITIVITY(SENSI,W,LAYER,MLAYR,NNODE,
+ MNODE,T,O,N,LL,MSAMP)
C
C   CALCULATE THE GRADIENT OF THE PERFORMANCE FUNCTION
C
C   CALL GRAD(SENSI,A,W,LAYER, MLAYR,
+ NNODE,MNODE,G,W0,LAMDA)
C
C   SUM UP THE TOTAL GRADIENT
C
C   CALL SUMGRAD(G,LAYER,MLAYR,NNODE,MNODE,TG)
320  CONTINUE
C
C   FIND THE PERFORMANCE FUNCTION VALUE, BEFORE LINE SEARCH
C
C   ERROR1= SQRT(FINDE(P,T,MSAMP,NSAMP,MNODE,

```

```

+ W,MLAYR,LAYER,NNODE,O,LAMDA,W0) /NSAMP)
  IF (MOD(ITER,PSTAT) .EQ. 1) THEN
    WRITE(*,1600)ITER,ERROR1
1600  FORMAT(1X,'BEFORE LINE SEARCH, ITER #',I5,2X,
+ 'ERROR1 VALUE = ',G25.20)
    ENDIF
C
C   FIRST START AND RESTART USING STEEPEST DESCENT
C
  IF (ITER .EQ. 1 .OR. MOD(ITER,NWEIG) .EQ. 0) THEN
    CALL GETPP(PP,PP0,TG,MLAYR,LAYER,MNODE,NNODE,
+ 0.D0)
    ENDIF
C
C ASSIGN THE TG TO TG0
C
  CALL ASSIGN(TG,TG0,MLAYR,LAYER,MNODE,NNODE)
  CALL ASSIGN(PP,PP0,MLAYR,LAYER,MNODE,NNODE)
C
C COMPUTE ALGORITHM 3.6.1 (4) AND (5).
  CALL LINMIN(FRET,P,T,MSAMP,NSAMP,
+ MNODE,W,PP0,MLAYR,LAYER,NNODE,O,LAMDA,W0)
C
C USING STEP 2 TO COMPUTE THE G(K+1)
C
  CALL INITG(LAYER,MLAYR,NNODE,MNODE,TG)
  DO 321 LL=1,NSAMP
C
C   FEEDFORWARD COMPUTATION
C
  CALL FORWARD(P,O,N,MLAYR,LAYER,MNODE,A,
+ NNODE,W,LL,MSAMP)
C
C CALCULATE THE SENSITIVITY MATRIX
C
  CALL SENSITIVITY(SENSI,W,LAYER,MLAYR,NNODE,
+ MNODE,T,O,N,LL,MSAMP)
C
C CALCULATE THE GRADIENT OF THE PERFORMANCE FUNCTION
C
  CALL GRAD(SENSI,A,W,LAYER,MLAYR,
+ NNODE,MNODE,G,W0,LAMDA)
C
C SUM UP THE TOTAL GRADIENT
C
  CALL SUMGRAD(G,LAYER,MLAYR,NNODE,MNODE,TG)
321  CONTINUE
C
C FIND THE PERFORMANCE FUNCTION VALUE, AFTER LINE SEARCH
C
  ERROR2= SQRT(FINDE(P,T,MSAMP,NSAMP,MNODE,
+ W,MLAYR,LAYER,NNODE,O,LAMDA,W0) /NSAMP)
C
  IF(MOD(ITER,PSTAT) .EQ. 0) THEN
    WRITE(*,1100)ITER,ERROR2

```

```

1100  FORMAT(1X,'AFTER LINE SEARCH, ITER# ',I5,2X,
+ 'ERROR2 VALUE = ',G25.20)
C    ENDIF

C    IF(MOD(ITER,PSTAT) .EQ. 1) THEN
C    IF (ABS(ERROR2 - ERROR1) .LT. TOL) THEN
        ERROR = ABS(ERROR2 - ERROR1)
        WRITE(*,101)ERROR
101   FORMAT (1X,'ERROR = ',G25.20)
C    STOP
C    ENDIF

C VALIDATE THE NETWORK USING VALIDATION SET.

C    IF(MOD(ITER,PSTAT) .EQ. 1) THEN
        ERROR = SQRT(TEST(MSAMP,MNODE,W,MLAYR,LAYER,
+ NNODE,LAMDA,W0)/NSAMP)
        WRITE(*,1200)ERROR
1200  FORMAT(1X,'AFTER TRAINING GENERALIZATION ERROR: ',G25.20)
C    ENDIF
C
        BETA=FINDBETA(TG,TG0,MLAYR,LAYER,MNODE,
+ NNODE)
        CALL GETPP(PP,PP0,TG,MLAYR,LAYER,MNODE,NNODE,
+ BETA)
C
C    PRINT TG, AFTER STEP 7
C
C ASSING TG TO TG0, TG0 STORES P(K+1)
C
        CALL ASSIGN(TG,TG0,MLAYR,LAYER,MNODE,NNODE)
C
        IF ((ABS(ERROR2 - ERROR1) .GT. TOL .OR. ERROR1 .GT. TOL1
+ .OR. ERROR2 .GT. TOL1) .AND. ITER .LT. MAXITER)THEN
            ERROR1 = ERROR2
            GOTO 1000
        ELSE
            IF (ITER .LT. MAXITER)THEN
                WRITE(*,1300)
1300   FORMAT(1X,'SOLUTION CONVERGE TO THE TOLERANCE')
            ENDIF

            WRITE(*,1400)ITER,ERROR1,ERROR2,ABS(ERROR2-ERROR1)
1400   FORMAT(1X,'ITER= ',I5,2X,'ERROR1= ',G25.10,2X,'ERROR2= ',
+ G25.10,2X,'ERROR=' G25.10)
C
C TEST THE NETWORK USING TEST SET
C
        ERROR = SQRT(TEST(MSAMP,MNODE,W,MLAYR,LAYER,
+ NNODE,LAMDA,W0)/NSAMP)
        WRITE(*,1500)ERROR
1500  FORMAT(1X,'AFTER TRAINING ERROR=',G25.10)
        ENDIF
C

```

```

STOP
END
C*****
SUBROUTINE GETINPUTDATA(P,T,MNODE,DIMIN,DIMOUT,MSAMP,
+ NSAMP)
C*****
C THIS SUBROUTINE IS TO READ THE INPUT DATA FROM *
C TRAINING SAMPLE AND THE TARGET OUTPUT DATA. *
C*****
INTEGER MNODE, DIMIN,DIMOUT,I,NSAMP,MSAMP,J
DOUBLE PRECISION P(MSAMP,MNODE), T(MSAMP,MNODE)
C
IN = 20
OPEN(UNIT = IN, FILE = 'TRAIN.DAT',STATUS = 'OLD',IOSTAT=IOERR)
IF(IOERR .NE. 0) THEN
WRITE(*,10) IOERR
10 FORMAT(1X,'CANNOT OPEN NETWORK TRAINING DATA FILE(TRAIN.DAT)',
+ I5)
STOP
ENDIF
C READ IN NUMBER OF TRAINING SAMPLE
READ(IN,*)NSAMP
C
DO 100 J= 1, NSAMP
C READ IN THE INPUT DATA
READ(IN,*)(P(J,I),I=1,DIMIN)
C
C READ IN THE DESIRED OUTPUT DATA(TARGET DATA)
READ(IN,*)(T(J,I),I=1,DIMOUT)
100 CONTINUE
C
CLOSE (UNIT=IN)
C
RETURN
END
C*****
SUBROUTINE PRINTINPUTDATA(P,T,MNODE,DIMIN,DIMOUT,MSAMP,
+ NSAMP)
C*****
C THIS SUBROUTINE IS TO PRINT THE INPUT DATA OF *
C TRAINING SAMPLE AND THE TARGET OUTPUT DATA. *
C*****
INTEGER MNODE, DIMIN,DIMOUT,I,NSAMP,MSAMP,J
DOUBLE PRECISION P(MSAMP,MNODE), T(MSAMP,MNODE)
C
C PRINT IN THE INPUT DATA
WRITE(*,100)NSAMP
100 FORMAT(1X,'NUMBER OF SAMPLE IS: ',I5)
C
DO 200 J=1,NSAMP
WRITE(*,300)J
300 FORMAT(1X,'SAMPLE # ',I5)
DO 20 I= 1,DIMIN
WRITE(*,400)P(I,J)

```



```

400     FORMAT(1X,'THE INPUT DATA ARE: '.E15.7)
20     CONTINUE
C
      DO 30 I= 1,DIMOUT
        WRITE(*,500)T(I,J)
500     FORMAT(1X,'THE DESIRED OUTPUT DATA ARE: '.E15.7)
30     CONTINUE
200    CONTINUE
C
      RETURN
      END
C*****
      SUBROUTINE ASSIGN(ORIG,NEW,MLAYR,NLAYR,MNODE,
+ NNODE)
C*****
C     THIS SUBROUTINE IS TO COPY A ORIG MATRIX TO NEW MATRIX.      *
C     IT IS USED TO COPY TG.                                       *
C*****
      INTEGER MLAYR,NLAYR,MNODE,NNODE(0:MLAYR)
      DOUBLE PRECISION ORIG(MLAYR,MNODE,0:MNODE),
+ NEW(MLAYR,MNODE,0:MNODE)
      INTEGER I,J,K,KK,LL
C
      DO 10 K=1,NLAYR
        KK=NNODE(K)
        LL=NNODE(K-1)
        DO 20 J=1,KK
          DO 30 I=0,LL
            NEW(K,J,I)=ORIG(K,J,I)
30     CONTINUE
20     CONTINUE
10     CONTINUE
C
      RETURN
      END
C*****
      FUNCTION BRENT(AX,BX,CX,F,TOL,XMIN)
C*****
C     GIVEN A FUNCTION F, AND GIVEN A BRACKETING      *
C     TRIPLET OF ABSCISSAS AX, BX, CX(SUCH THAT BX IS  *
C     BETWEEN AX, AND CX, AND F(BX) IS LESS THAN BOTH  *
C     F(AX) AND F(CX)), THIS ROUTINE ISOLATES THE MINIMUM  *
C     TO A FRACTIONAL PRECISION OF ABOUT TOL USING BRENT'S  *
C     METHOD. THIS ABXCISSA OF THE MINIMUM IS RETURNED AS  *
C     XMIN, AND MUNIMUM FUNCTION VALUE IS RETURNED AS BRENT.  *
C     THE RETURNED FUNCTION VALUE.                       *
C     *
C     PARAMETERS. MAXIMUM ALLOWED NUMBER OF ITERATIONS:GOLDEN*
C     RATIO, AND A SMALL NUMBER THAT PROTECTS AGAINST TRYING *
C     TO ACHIEVE FRACTION ACCURACY FOR A MINIMUM THAT HAPPENS *
C     TO BE EXACTLY ZERO.                                 *
C*****
      INTEGER ITMAX
      DOUBLE PRECISION BRENT, AX,BX,CX,TOL,XMIN,F,CGOLD,ZEPS

```

```

EXTERNAL F
  PARAMETER(ITMAX=100, CGOLD=.381966D0,ZEPS=1.0D-10)
C
  INTEGER ITER
  DOUBLE PRECISION A,B,D,E,ETEMP,FU,FV,FW,FX,P,Q,R,TOL1,TOL2.
+ U,V,W,X,XM
  A=MIN(AX,CX)
  B=MAX(AX,CX)
  V=BX
  W=V
  X=V
  E=0.D0
  FX=F(X)
  FV=FX
  FW=FX
  DO 11 ITER = 1, ITMAX
    XM = .5D0*(A+B)
    TOL1 = TOL*ABS(X) + ZEPS
    TOL2 = 2.D0*TOL1
    IF(ABS(X-XM) .LE. (TOL2 - .5D0*(B-A))) GOTO 3
    IF(ABS(E) .GT. TOL1)THEN
      R=(X-W)*(FX-FV)
      Q=(X-V)*(FX-FW)
      P=(X-V)*Q-(X-W)*R
      Q=2.D0*(Q-R)
      IF(Q.GT.0) P=-P
      Q=ABS(Q)
      ETEMP=E
      E=D
      IF(ABS(P).GE.ABS(.5D0*Q*ETEMP).OR.P.LE.Q*(A-X).OR.
+      P.GE.Q*(B-X))GOTO 1
      D=P/Q
      U=X+D
      IF(U-A.LT.TOL2 .OR. B-U .LT. TOL2)D=DSIGN(TOL1,XM-X)
      GOTO 2
    ENDIF
1  IF(X.GE.XM)THEN
    E=A-X
  ELSE
    E=B-X
  ENDIF
  D=CGOLD*E
2  IF(ABS(D).GE.TOL1)THEN
    U=X+D
  ELSE
    U=X+DSIGN(TOL1,D)
  ENDIF
  FU = F(U)
  IF(FU.LE.FX)THEN
    IF(U.GE.X)THEN
      A=X
    ELSE
      B=X
    ENDIF
  ENDIF

```

```

      V=W
      FV=FW
      W=X
      FW=FX
      X=U
      FX=FU
    ELSE
      IF(U.LT.X)THEN
        A=U
      ELSE
        B=U
      ENDIF
      IF(FU.LE.FW .OR. W.EQ.X)THEN
        V=W
        FV=FW
        W=U
        FW=FU
      ELSEIF(FU .LE. FV .OR. V.EQ.X .OR. V.EQ.W)THEN
        V=U
        FV=FU
      ENDIF
    ENDIF
11  CONTINUE
C
3   XMIN=X
    BRENT=FX
    RETURN
    END
C*****
C   SUBROUTINE CONVERT(TG,MLAYR,MNODE,NLAYR,
+   NNODE,A,MAXNUM,NUM)
C*****
C   THIS ROUTINE IS TO CONVERT THE 3-DIMENSIONAL      *
C   ARRAYS INTO 1-DIMENSIONAL ARRAY. IT IS USED      *
C   TO APPLY LINE SEARCH ROUTINE                      *
C*****
C   INTEGER MLAYR,MNODE,NLAYR,NNODE(0:MLAYR),
+   MAXNUM,NUM,I,J,K,KK,LL
C   DOUBLE PRECISION A(MAXNUM),TG(MLAYR,MNODE,0:MNODE)
C
C   NUM=0
C   DO 10 K=1,NLAYR
C     KK=NNODE(K)
C     LL=NNODE(K-1)
C     DO 20 J=1,KK
C       DO 30 I=1,LL
C         NUM=NUM+1
C         A(NUM)=TG(K,J,I)
30      CONTINUE
20    CONTINUE
10   CONTINUE
C
    RETURN
    END

```

```

C*****
C*****
C   GIVEN A FUNCTION F AND ITS DERIVATIVE FUNCTION DF, AND   *
C   GIVEN A BRACKETING TRIPLET OF ABSCISSAS AX, BX, CX[SUCH  *
C   THAT BX IS BETWEEN AX AND CX AND F(BX) IS LESS THAN BOTH *
C   F(AX) AND F(CX)], THIS ROUTINE ISOLATES THE MINIMUM TO A  *
C   FRACTIONAL PRECISION OF ABOUT TOL USING A MODIFICATION OF *
C   BRENT'S METHOD THAT USES DERIVATIVES. THE ABSCISSA OF THE *
C   MINIMUM IS RETURNED AS XMIN, AND THE MINIMUM FUNCTION    *
C   VALUE IS RETURNED AS DBRENT, THE RETURNED FUNCTION VALUE.*
C*****
      FUNCTION DBRENT(AX,BX,CX,F,DF,TOL,XMIN)
      INTEGER ITMAX
      DOUBLE PRECISION DBRENT,AX,BX,CX,TOL,XMIN,DF,F,ZEPS
      EXTERNAL DF,F
      PARAMETER(ITEM=100,ZEPS=1.0D-10)
      INTEGER ITER
      DOUBLE PRECISION A,B,D,D1,D2,DU,DV,DW,DX,E,FU,FV,FW,FX,OLDE,
+ TOL1, TOL2, U,U1,U2,V,W,X,XM
      LOGICAL OK1,OK2
      A=MIN(AX,CX)
      B=MAX(AX,CX)
      V=BX
      W=V
      X=V
      E=0.
      FX=F(X)
      FV=FX
      FW=FX
      DX=DF(X)
      DV=DX
      DW=DX
      DO 11 ITER=1,ITMAX
         XM=0.5*(A+B)
         TOL1=TOL*ABS(X)+ZEPS
         TOL2=2.*TOL1
         IF(ABS(X-XM) .LE. (TOL2 - .5*(B-A)))GOTO 3
         IF(ABS(E) .GT. TOL1) THEN
            D1=2.*(B-A)
            D2=D1
            IF(DW.NE.DX)D1=(W-X)*DX/(DX-DW)
            IF(DV.NE.DX)D2=(V-X)*DX/(DX-DV)
            U1=X+D1
            U2=X+D2
            OK1=((A-U1)*(U1-B).GT.0) .AND. (DX*D1 .LE. 0.)
            OK2=((A-U2)*(U2-B).GT.0) .AND. (DX*D2 .LE. 0.)
            OLDE=E
            E=D
            IF(.NOT.(OK1.OR.OK2))THEN
               GOTO 1
            ELSEIF (OK1 .AND. OK2)THEN
               IF(ABS(D1).LT.ABS(D2))THEN
                  D=D1
               ELSE

```

```

        D=D2
    ENDIF
ELSEIF (OK1) THEN
    D=D1
ELSE
    D=D2
ENDIF
IF(ABS(D) .GT. ABS(0.5*OLDE))GOTO 1
U=X+D
IF(U-A .LT. TOL2 .OR. B-U .LT. TOL2)D=SIGN(TOL1,XM-X)
GOTO 2
ENDIF
1 IF(DX.GE.0.)THEN
    E=A-X
ELSE
    E=B-X
ENDIF
D=.5*E
2 IF(ABS(D) .GE. TOL1)THEN
    U=X+D
    FU=F(U)
ELSE
    U=X+SIGN(TOL1,D)
    FU=F(U)
    IF(FU.GT.FX)GOTO 3
ENDIF
DU=DF(U)
IF(FU.LE.FX)THEN
    IF(U.GE.X) THEN
        A=X
    ELSE
        B=X
    ENDIF
    V=W
    FV=FW
    DV=DW
    W=X
    FW=FX
    DW=DX
    X=U
    FX=FU
    DX=DU
ELSE
    IF(U.LT.X)THEN
        A=U
    ELSE
        B=U
    ENDIF
    IF(FU.LE.FW .OR. W.EQ.X)THEN
        V=W
        FV=FW
        DV=DW
        W=U
        FW=FU

```

```

                                DW=DU
                                ELSEIF(FU .LE. FV .OR. V.EQ.X .OR. V.EQ.W)THEN
                                    V=U
                                    FV=FU
                                    DV=DU
                                ENDF
                                ENDF
11    CONTINUE
3     XMIN=X
      DBRENT=FX
      RETURN
      END

```

```

      FUNCTION FINDBETA(TG,TG0,MLAYR,NLAYR,MNODE,
+ NNODE)
C*****
C   THIS ROUTINE IS TO FIND THE BETA ACCORDING TO      *
C   (2.4.8) -- (2.4.10).                               *
C   TG(MLAYR, MNODE, 0:MNODE) STORES TOTAL          *
C   GRADIENT                                           *
C*****
      INTEGER MLAYR,NLAYR,MNODE,NNODE(0:MLAYR)
      DOUBLE PRECISION TG(MLAYR,MNODE,0:MNODE),
+ TG0(MLAYR,MNODE,0:MNODE),FINDBETA,SUM,SUM1
      INTEGER I,J,K,LL
C
      SUM=0.D0
      SUM1=0.D0
      DO 10 K=1,NLAYR
        KK=NNODE(K)
        LL=NNODE(K-1)
        DO 20 J=1,KK
          DO 30 I=0,LL
            SUM = SUM + TG(K,J,I)*TG(K,J,I)
            SUM1 = SUM1 + TG0(K,J,I)*TG0(K,J,I)
          30    CONTINUE
        20    CONTINUE
      10    CONTINUE
C
      FINDBETA=SUM/SUM1
      RETURN
      END

```

```

C*****
      FUNCTION FINDE(P,T,MSAMP,NSAMP,MNODE,
+ W,MLAYR,NLAYR,NNODE,O,LAMDA,W0)
C*****
C   THIS FUNCTION IS TO FIND THE PERFORMANCE          *
C   FUNCTION E(W) REF. (3.6.1).                       *
C   FINDE -- THE PERFORMANCE VALUE. REF (3.6.1)       *
C   T(MSAMP,MNODE)-- THE DESIRED OUTPUT OF THE NET  *
C   W(MLAYR,MNODE,0:MNODE)--WEIGHT MATRIX OF THE NET *
C   O(MSAMP,MNODE)-- THE CALCULATED OUTPUT OF THE NET*
C   LAMDA-- THE CONSTANT IN THE PENALTY TERM         *
C*****

```

```

C      W0 -- THE CONSTANTS IN THE PENALTY.
C*****
C
C      INTEGER MSAMP,NSAMP,MNODE,MLAYR,NLAYR,
+      NNODE(0:MLAYR)
C      DOUBLE PRECISION O(MSAMP,MNODE),T(MSAMP,MNODE),
+      W(MLAYR,MNODE,0:MNODE),LAMDA,W0,SUM,SUM1,FINDE,
+      P(MSAMP,MNODE)
C      DOUBLE PRECISION N(MLAYR,MNODE),A(0:MLAYR,0:MNODE)
C      INTEGER I,J,K,L,KK,LL
C
C      CALCULATE THE PENALTY TERM.
C
C      SUM=0.D0
C      SUM1=0.D0
C      DO 100 K=1,NLAYR
C      KK=NNODE(K)
C      LL=NNODE(K-1)
C      DO 200 J=1,KK
C      DO 300 I=0,LL
C      SUM=SUM+LAMDA*(W(K,J,I)**2/(W0**2+W(K,J,I)**2))
300      CONTINUE
200      CONTINUE
100      CONTINUE
C
C      CALCULATE THE FIRST TERM
C
C      DO 10 L=1,NSAMP
C      CALL FORWARD(P,O,N,MLAYR,NLAYR,MNODE,A,NNODE,
+      W,L,MSAMP)
C      DO 20 K=1,NNODE(NLAYR)
C      SUM1=SUM1+(T(L,K)-O(L,K))**2
20      CONTINUE
10      CONTINUE
C
C      FINDE = 0.5D0* (SUM +SUM1)
C
C      RETURN
C      END
C*****
C      SUBROUTINE FORWARD(P,O,N,MLAYR,NLAYR,MNODE,A,
+      NNODE,W,SN,MSAMP)
C*****
C      THIS SUBROUTINE IS TO CALCULATE THE SUM OF
C      THE INPUTS OF A NEURON J IN LAYER K
C      PLEASE REFER TO (3.1.1)
C      N(MLAYR,MNODE)--STORES THE SUM OF INPUTS OF
C      NEURON J IN LAYER K
C      A(0:MLAYR,MNODE)--STORES THE OUTPUT OF
C      NEURON J IN LAYER K
C      A(0:MLAYR,0:MNODE) -- STORES THE INPUT DATA.
C      P(MSAMP,MNODE) -- IS THE INPUT DATA FROM ONE SAMPLE
C      T(MSAMP,MNODE) -- IS THE DESIRED OUTPUT DATA FROM ONE
C      SAMPLE

```

```

C      O(MSAMP,MNODE) -- IS THE OUTPUT CALCULATED FROM THE      *
C                               NET.                               *
C      W(MLAYR,MNODE,0:MNODE) -- THE WEIGHT OF THE NET.        *
C      SN -- THE SAMPLE INDEX.                                  *
C*****
      INTEGER MLAYR,MNODE,NNODE(0:MLAYR),I,J,K,
+     NLAYR,L,SN,MSAMP,KK,LL
      DOUBLE PRECISION N(MLAYR,MNODE),A(0:MLAYR,0:MNODE),
+     W(MLAYR,MNODE,0:MNODE),SUM,P(MSAMP,MNODE),
+     O(MSAMP,MNODE)
C
C      STORE INPUT DATA INTO A(0,MNODE)
C
      DO 100 I=1, NNODE(0)
        A(0,I)=P(SN,I)
100  CONTINUE
C
C STORE THE BIAS
C
      A(0,0) = -1.D0
C
C      CALCULATE THE SUM OF THE INPUTS OF A NEURON J IN LAYER K
C
C      LOOP OVER LAYER
C LOOP OVER LAYER
      DO 10 K=1,NLAYR
C LOOP OVER CURRENT NODE (TARGET)
        KK=NNODE(K)
        LL=NNODE(K-1)
        DO 20 J=1,KK
C LOOP OVER PREVIOUS NODE (SOURCE)
          SUM = 0.0D0
          DO 30 I=0,LL
            SUM = SUM + W(K,J,I)*A(K-1,I)
30      CONTINUE
C
C CALCULATE THE SUM OF I NEURON J IN LAYER K
C
        N(K,J) = SUM
C
C CALCULATE THE OUTPUT OF NEURON J IN LAYER K
C
        A(K,J) = SIGF(N(K,J))
20      CONTINUE
C
C THE BIAS
C
      A(K,0) = -1
10      CONTINUE
C
C      STORED THE OUTPUT IN A(NNODE(NLAYR))
C
      KK=NNODE(NLAYR)
      DO 200 I=1, KK

```



```

      O(SN,I)=A(NLAYR,I)
200  CONTINUE
      RETURN
      END
C*****
      SUBROUTINE FRPRMN(P,N,FTOL,ITER,FRET)
C*****
C      GIVEN A STARTING POINT P THAT IS A VECTOR OF LENGTH          *
C      N, FLETCH-REEVES-POLAK-RIBIERE MINIMIZATION IS              *
C      PERFORMED ON A FUNCTION FUNC, USING ITS GRADIENT AS        *
C      CALCULATED BY A ROUTINE DFUNC. THE CONVERGENCE TOLERANCE   *
C      ON THE FUNCTION VALUE IS INPUT AS FTOL. RETURNED          *
C      QUANTITIES ARE P(THE LOCATION OF THE MINIMUM), ITER(THE    *
C      NUMBER OF ITERATIONS THAT WERE PERFORMED), AND FRET(THE    *
C      MINIMUM VALUE OF THE FUNCTION). THE ROUTINE LINMIN IS      *
C      CALLED TO PERFORM LINE MINIMIZATIONS.                      *
C      PARAMETERS: NMAX IS THE MAXIMUM ANTICIPATED VALUE OF N;    *
C      ITMAX IS THE MAXIMUM ALLOWED NUMBER OF ITERATIONS; EPS     *
C      IS A SMALL NUMBER TO RECTIFY SPECIAL CASE OF CONVERGING   *
C      TO EXACTLY ZERO FUNCTION VALUE.                             *
C*****
      INTEGER ITER,N,NMAX,ITMAX
      DOUBLE PRECISION FRET,FTOL,P(N),EPS,FUNC
      EXTERNAL FUNC
      PARAMETER(NMAX=50,ITMAX=200,EPS=1.0D-10)
C
C      USES DFUNC,FUNC,LINMIN
C
      INTEGER ITS,J
      DOUBLE PRECISION DGG,GAM,GG,G(NMAX),H(NMAX),XI(NMAX)
      FP = FUNC(P)
      CALL DFUNC(P,XI)
      DO 11 J=1,N
         G(J)=XI(J)
         H(J)=G(J)
         XI(J)=H(J)
11     CONTINUE
      DO 14 ITS=1,ITMAX
         CALL LINMIN(P,XI,N,FRET)
         IF(2.*ABS(FRET-FP) .LE. FTOL*(ABS(FRET)+ABS(FP)+EPS))RETURN
         FP = FUNC(P)
         CALL DFUNC(P,XI)
         GG =0.D0
         DGG =0.D0
         DO 12 J=1,N
            GG=GG+G(J)**2
            DGG=DGG+(XI(J)+G(J))*XI(J)
12     CONTINUE
         IF(GG .EQ. 0)RETURN
         GAM=DGG/GG
         DO 13 J=1,N
            G(J)=XI(J)
            H(J)=G(J)+GAM*H(J)
            XI(J)=H(J)

```

```

13 CONTINUE
14 CONTINUE
C
    RETURN
    END
C*****
SUBROUTINE GETPP(PP,PP0,TG,MLAYR,NLAYR,MNODE,
+ NNODE, BETA)
C*****
C THIS SUBROUTINE IS TO CALCULATE METRIX PP. REF. *
C ALGORITHM 3.6.1 (3) AND (7). IT ADDS THE PREVIOUS *
C GRADIENT TO THE CURRENT GRADIENT ACCORDING TO *
C DIFFERENT BETA. REF.(2.4.8)-(2.4.10). STORED THE *
C WHOLE GRADIENT IN PP. *
C*****
INTEGER MLAYR,NLAYR,MNODE,NNODE(0:MLAYR)
DOUBLE PRECISION PP(MLAYR,MNODE,0:MNODE),BETA,
+ TG(MLAYR,MNODE,0:MNODE),
+ PP0(MLAYR,MNODE,0:MNODE)
INTEGER I,J,K,KK,LL
C
C CALCULATE THE GRADIENT AND STORE IT IN PP
C
    DO 10 K=1,NLAYR
    KK=NNODE(K)
    LL=NNODE(K-1)
    DO 20 J=1,KK
        DO 30 I=0,LL
            PP(K,J,I) = -TG(K,J,I) + BETA * PP0(K,J,I)
30 CONTINUE
20 CONTINUE
10 CONTINUE
C
    RETURN
    END
C*****
SUBROUTINE GRAD(SENSI,A,W,NLAYR, MLAYR,
+ NNODE,MNODE,G,W0,LAMDA)
C*****
C THIS SUBROUTINE IS TO CALCULATE THE *
C GRADIENT OF THE PERFORMANCE W.R.T WEIGHT *
C REF. (3.6.6). *
C SENSI(MLAYR,MNODE)--THE SENSITVITY MATRIX. REF(3.6.12) *
C A(0:MLAYR,0:MNODE) -- THE OUTPUT OF A NEURON REF(3.1.2) *
C W(MLAYR,MNODE,0:MNODE)-- THE WEIGHT MATRIX *
C G(MLAYR,MNODE,0:MNODE)-- THE GRADIENT OF THE NET *
C OF ONE SAMPLE DATE. *
C W0 -- THE CONSTANTS IN PENALTY TERM W0. *
C LAMDA -- THE CONSTANT IN THE PENALTY *
C*****
INTEGER NLAYR, MLAYR,MNODE,I,J,K,KK,LL,
+ NNODE(0:MLAYR)
DOUBLE PRECISION SENSI(MLAYR,MNODE),A(0:MLAYR,
+ 0:MNODE),W(MLAYR,MNODE,0:MNODE),

```

```

+ G(MLAYR,MNODE,0:MNODE),W0,LAMDA
C
C   CALCULATE THE GRADIENT OF PERFORMACE FUNCTION W.R.T
C   WEIGHTS ACCORDING TO (3.6.6)
C
C   LOOP OVER LAYER
C
      DO 10 K=1,NLAYR
      KK=NNODE(K)
      LL=NNODE(K-1)
      DO 20 J=1,KK
C
C   BIAS TERM
C
      G(K,J,0)=SENSI(K,J)+
+      LAMDA * (W(K,J,0) * W0*W0)/(W0*W0 + W(K,J,0)**2)
          DO 30 I=1,LL
          G(K,J,I)=SENSI(K,J) * A(K-1,I) +
+          LAMDA * (W(K,J,I) * W0*W0)/(W0*W0 + W(K,J,I)**2)
30      CONTINUE
20      CONTINUE
10      CONTINUE
C
      RETURN
      END
C*****
      SUBROUTINE INITG(NLAYR,MLAYR,NNODE,
+ MNODE,TG)
C*****
C   THIS FUNCTION IS TO INITIALIZE THE TOTAL      *
C   GRADIENT TO 0.                                *
C*****
      INTEGER NLAYR, MLAYR,MNODE,I,J,K,KK,LL,
+ NNODE(0:MLAYR)
      DOUBLE PRECISION TG(MLAYR,MNODE,0:MNODE)
C
C   INITIALIZE THE TOTAL GRADIENT TO 0
C   AND NUMOFSAMPLE TO 0
C
      DO 10 K=1,NLAYR
      KK=NNODE(K)
      LL=NNODE(K-1)
      DO 20 J=1,KK
          DO 30 I=0,LL
          TG(K,J,I)= 0
30      CONTINUE
20      CONTINUE
10      CONTINUE
C
      RETURN
      END
C*****
      SUBROUTINE INIWEIGHT(WEIGHT, NLAYR, MLAYR,NNODE,
+ MNODE, SEED,NWEIG)

```

```

C*****
C   INITIALIZE THE WEIGHT OF INPUT LAYER      *
C                                           *
C   NLAYR -- THE NUMBER OF LAYER (INCLUDING  *
C           OUTPUT AND HIDDEN LAYERS).      *
C                                           *
C   NUMNODE(I) -- THE NUMBER OF NODE AT LAYER I. *
C   NUMNODE(0) -- THE NUMBER OF INPUT (NODE). *
C   NUMNODE(NLAYR) -- NUMBER OF NODE IN OUTPUT LAYER *
C   NWEIG -- THE NUMBER OF WEIGHT          *
C                                           *
C   WEIGHT(LAYER, N, 0:N)-- LAYER IN THE LAYER INDEX *
C           N,M CORRESPONDING TO W(J,I), I.E., *
C           WEIGHT(LAYER, N, M) IS THE WEIGHT *
C           OF THE CONNECTION FROM NODE M OF *
C           THE (LAYER-1)TH LAYER TO NODE N OF *
C           THE LAYERTH LAYER. *
C           WEIGHT(LAYER, N, 0) IS THE BIAS. *
C                                           *
C*****
      INTEGER MLAYR, MNODE,NWEIG
      INTEGER I,J,K,FANIN,NNODE(0:MLAYR),NLAYR,KK,LL
      DOUBLE PRECISION WEIGHT(MLAYR, MNODE,0:MNODE), TEMP,
+ DRANDOM,SEED,TEMP1
C
C GENERATE THE RANDOM NUMBER BETWEEN -0.5 TO 0.5
C
      TEMP1 = SEED
      TEMP = DRANDOM(TEMP1) - .5D0
      NWEIG = 0
C
C LOOP OVER LAYER
C
      DO 10 K=1, NLAYR
C
C CALCULATE THE FAN-IN OF THE LAYER.
C
      KK=NNODE(K)
      LL=NNODE(K-1)
      FANIN = NNODE(K-1) + 1
C
C LOOP OVER ALL NEURONS IN CURRENT LAYER
C
      DO 20 J=1, KK
C
C LOOP OVER ALL NEURONS IN PREVIOUS LAYER
C
      DO 30 I = 0, LL
          WEIGHT(K,J,I) = TEMP/FANIN
          NWEIG = NWEIG + 1
30      CONTINUE
20      CONTINUE
10      CONTINUE
      RETURN

```

```

      END
C*****
      SUBROUTINE LINMIN(FRET,P,T,MSAMP,NSAMP,
+ MNODE,W,TG,MLAYR,NLAYR,NNODE,O,LAMDA,W0)
C*****
C   GIVEN AN N-DIMENSIONAL POINT P(1:N) AND AN      *
C   N-DIMENSIONAL DIRECTION XI(1:N), MOVES AND      *
C   RESETS P TO WHERE THE FUNCTION FUNC(P) TAKES ON *
C   A MINIMUM ALONG THE DIRECTION XI FROM P AND     *
C   REPLACES XI BY THE ACTUAL VECTOR DISPLACEMENT  *
C   THAT P WAS MOVED ALSO RETURNS AS FRET THE VALUE *
C   OF FUNC AT THE RETURNED LOCATION P. THIS IS    *
C   ACTUALLY ALL ACCOMPLISHED BY CALLING THE ROUTINES*
C   MNBRK AND BRENT.                                *
C                                                    *
C   REF "NUMERICAL RECIPIES"                        *
C*****
      INTEGER MSAMP,NSAMP,MNODE,MLAYR,NLAYR,
+ NNODE(0:MLAYR)
      DOUBLE PRECISION O(MSAMP,MNODE),T(MSAMP,MNODE),
+ W(MLAYR,MNODE,0:MNODE),TG(MLAYR,MNODE,0:MNODE),
+ P(MSAMP,MNODE),LAMDA,W0,TOL,FRET
      INTEGER MSCOM,NSCOM,MNCOM,MLCOM,
+ NLCOM
      PARAMETER(MLCOM=4,MNCOM=100,MSCOM=200)
      INTEGER NNCOM(0:MLCOM)
      DOUBLE PRECISION OCOM(MSCOM,MNCOM),TCOM(MSCOM,
+ MNCOM),WCOM(MLCOM,MNCOM,0:MNCOM),
+ LAMCOM,W0COM,TGCOM(MLCOM,MNCOM,0:MNCOM),
+ PCOM(MSCOM,MNCOM)
      DOUBLE PRECISION AX,BX,FA,FB,FX,XMIN,XX,BRENT
      COMMON /F1/NSCOM, NLCOM, NNCOM
      COMMON /F2/PCOM,OCOM,TCOM,WCOM,TGCOM,LAMCOM,W0COM
      INTEGER I,J,K,KK,LL
      EXTERNAL FIDIM
C
C INITIALIZE THE PARAMETERS
C
      NSCOM=NSAMP
      NLCOM=NLAYR
      LAMCOM=LAMDA
      W0COM=W0
      TOL=1.0D-4
      DO 10 I=0,NLCOM
         NNCOM(I)=NNODE(I)
10  CONTINUE
      DO 50 I=1,NSCOM
         KK=NNCOM(NLCOM)
         DO 60 J=1,KK
            OCOM(I,J)=O(I,J)
            TCOM(I,J)=T(I,J)
            PCOM(I,J)=P(I,J)
60  CONTINUE
50  CONTINUE

```

```

C
DO 20 K=1,NLCOM
  KK=NNODE(K)
  LL=NNODE(K-1)
  DO 30 J=1,KK
    DO 40 I=0,LL
      WCOM(K,J,I)=W(K,J,I)
      TGCOM(K,J,I)=TG(K,J,I)
40    CONTINUE
30  CONTINUE
20  CONTINUE
C
C   USES BRENT,F1DIM,MNBRAK
C
  AX = -1.0D0
  XX = 1.0D0
  CALL MNBRAK(AX,XX,BX,FA,FX,FB,F1DIM)
  FRET = BRENT(AX,XX,BX,F1DIM,TOL,XMIN)
C
C CALCULATE THE TOTAL GRADIENT
C
DO 70 K=1,NLCOM
  KK=NNODE(K)
  LL=NNODE(K-1)
  DO 80 J=1,KK
    DO 90 I=0,LL
      TG(K,J,I)=XMIN*TG(K,J,I)
      W(K,J,I)=W(K,J,I)+TG(K,J,I)
90    CONTINUE
80  CONTINUE
70  CONTINUE
C
  RETURN
  END

C*****
  FUNCTION F1DIM(X)
  INTEGER MSCOM,NSCOM,MNCOM,MLCOM,
+ NLCOM
  PARAMETER(MLCOM=4,MNCOM=100,MSCOM=200)
  INTEGER NNCOM(0:MLCOM)
  DOUBLE PRECISION OCOM(MSCOM,MNCOM),TCOM(MSCOM,
+ MNCOM),WCOM(MLCOM,MNCOM,0:MNCOM),
+ LAMCOM,W0COM,TGCOM(MLCOM,MNCOM,0:MNCOM),
+ PCOM(MSCOM,MNCOM)
  DOUBLE PRECISION XT(MLCOM,MNCOM,0:MNCOM)
C
C THE COMMON BLOCK
C
COMMON /F1/NSCOM, NLCOM, NNCOM
COMMON /F2/PCOM,OCOM,TCOM,WCOM,TGCOM,LAMCOM,W0COM
DOUBLE PRECISION F1DIM,X
EXTERNAL FINDE
C

```

```

C USES FINDE
C   USED BY LINMIN AS THE FUNCTION PASSED MNBRAK AND BRENT
C
C   INTEGER I,J,K,KK,LL
C   DO 100 K=1,NLCOM
C     KK=NNCOM(K)
C     LL=NNCOM(K-1)
C     DO 200 J=1,KK
C       DO 300 I=0,LL
C         XT(K,J,I)=WCOM(K,J,I)+X*TGCOM(K,J,I)
300   CONTINUE
200   CONTINUE
100   CONTINUE
      F1DIM = FINDE(PCOM,TCOM,MSCOM,NSCOM,MNCOM,XT,
+ MLCOM,NLCOM,NNCOM,OCOM,LAMCOM,W0COM)
      RETURN
      END
C*****
C   SUBROUTINE MNBRAK(AX,BX,CX,FA,FB,FC,FUNC)
C*****
C   THIS ROUTINE IS TO INITIALLY BRACKETING          *
C   A MINIMUM. REF " NUMERICAL RECIPIES            *
C   IN FORTRAN, THE ART OF SCIENTIFIC COMPUTING"    *
C   BY WILLIAM H. PRESS, ETC.                       *
C                                                    *
C   GIVEN A FUNCTION FUNC AND GIVEN DISTINCT        *
C   INITIAL POINTS AX AND BX, THIS ROUTINE          *
C   SEARCHES IN THE DOWNHILL DIRECTION (DEFINED     *
C   BY THE FUNCTION AS EVALUATED AT THE INITIAL    *
C   POINTS) AND RETURNS NEW POINTS AX, BX,         *
C   CX THAT BRACKET A MINIMUM OF THE FUNCTION      *
C   ALSO RETURNED ARE THE FUNCTION VALUES AT     *
C   THE THREE POINTS, FA, FB AND FC.              *
C   PARAMETERS: GOLD IS THE DEFAULT RATIO BY       *
C   WHICH SUCCESSIVE INTERVALS ARE MAGNIFIED.     *
C   GLIMIT IS THE MAXIMUM MAGNIFICATION FOR        *
C   A PARABOLIC-FIT STEP.                          *
C*****
C   DOUBLE PRECISION AX,BX,CX,FA,FB,FC,FUNC,GOLD,GLIMIT,TINY
C   EXTERNAL FUNC
C   PARAMETER (GOLD=1.618034D0,GLIMIT=100.D0,TINY=1.D-20)
C   DOUBLE PRECISION DUM,FU,Q,R,U,ULIM
C   FA=FUNC(AX)
C   FB=FUNC(BX)
C   IF(FB .GT. FA) THEN
C     DUM=AX
C     AX=BX
C     BX=DUM
C     DUM=FB
C     FB=FA
C     FA=DUM
C   ENDIF
C
C   FIRST GUESS FOR C

```

```

C      CX = BX +GOLD*(BX-AX)
      FC = FUNC(CX)
C
C INITIALIZE THE ITERATION COUNT
C
      ITER=0
1  IF(FB.GE.FC)THEN
      R=(BX-AX)*(FB-FC)
      Q=(BX-CX)*(FB-FA)
      U=BX-((BX-CX)*Q-(BX-AX)*R)/(2.*SIGN(MAX(ABS(Q-R),
+    TINY),Q-R))
      ULIM=BX + GLIMIT *(CX-BX)
      IF((BX-U)*(U-CX) .GT. 0) THEN
      FU = FUNC(U)
      IF(FU .LT. FC)THEN
      AX = BX
      FA = FB
      BX = U
      FB = FU
      RETURN
      ELSE IF(FU .GT. FB) THEN
      CX = U
      FC = FU
      RETURN
      ENDIF
      U = CX +GOLD*(CX - BX)
      FU = FUNC(U)
      ELSE IF((CX-U)*(U-ULIM) GT.0)THEN
      FU = FUNC(U)
      IF(FU .LT. FC) THEN
      BX = CX
      CX = U
      U = CX + GOLD*(CX - BX)
      FB = FC
      FC = FU
      FU = FUNC(U)
      ENDIF
      ELSE IF((U - ULIM)*(ULIM - CX) .GE. 0)THEN
      U = ULIM
      FU = FUNC(U)
      ELSE
      U = CX + GOLD * (CX - BX)
      FU = FUNC(U)
      ENDIF
      AX = BX
      BX = CX
      CX = U
      FA = FB
      FB = FC
      FC = FU
      ITER = ITER +1
      GO TO 1
      ENDIF

```



```

RETURN
END
*****
      SUBROUTINE NETPRINT(LAYER,MLAYR,NNODE,SEED,W0,
+      LAMDA,METHOD)
C*****
C      THIS SUBROUTINE IS TO PRINT THE NETWORK ARCHITCTURE AND *
C      INITIAL PARAMETERS. *
C*****
      INTEGER LAYER,MLAYR,NNODE(0:MLAYR),METHOD,NSAMP
      DOUBLE PRECISION SEED, TOL, W0, LAMDA
C
      WRITE(*,10)LAYER
10  FORMAT(1X,'THE NUMBER OF LAYER IN THE NETWORK IS: ',I4)
      WRITE(*,20)NNODE(0)
20  FORMAT(1X,'THE INPUT DIMENSION IS ', I4)
      DO 30 I=1, LAYER
          WRITE(*,40)I,NNODE(I)
40  FORMAT(1X,'THE NUMBER OF NODE IN LAYER ',I4,' IS ', I4)
30  CONTINUE
      WRITE(*,60)NNODE(LAYER)
60  FORMAT(1X,'THE OUTPUT DIMENSION IS ', I4)
      IF (METHOD .EQ. 0) THEN
          WRITE(*,100)
100  FORMAT(1X,'THE PENALTY METHOD IS USED')
      ELSE IF(METHOD .EQ. 1) THEN
          WRITE(*,200)
200  FORMAT(1X,'THE STOP TRAINING METHOD IS USED')
      ELSE
          WRITE(*,300)
300  FORMAT(1X,'METHOD DATA ERROR')
          STOP
      ENDIF
C      PRINT THE PARAMETERS
      WRITE(*,50)SEED,W0,LAMDA
50  FORMAT(1X,'THE SEED IS ',F10.4/1X,
+ /1X,'THE W0 IS ',F16.12/1X,'THE LAMDA IS ', F16.12)
C
      RETURN
      END
C*****
      SUBROUTINE NETSETUP(LAYER,MLAYR,NNODE,SEED,
+      W0,LAMDA,METHOD)
C*****
C      *
C      THIS SUBROUTINE IS TO READ THE INPUT FILE AND SET UP *
C      THE NETWORK ARCHITECTURE AND INITIALIZE PARAMETERS*
C      *
C*****
      INTEGER LAYER, MLAYR, MAXNODE, NNODE(0:MLAYR),METHOD,
+      NSAMP
      DOUBLE PRECISION SEED,TOL,W0,LAMDA
C
      IN=50

```

```

OPEN(UNIT = IN, FILE = 'NET.DAT', STATUS = 'OLD', IOSTAT= IOERR)
IF(IOERR .NE. 0) THEN
WRITE(*,10)IOERR
10  FORMAT('CANNOT OPEN NETWORK DATA FILE (NET.DAT), IOERR= ',I10)
      STOP
ENDIF
C
C  READ IN THE NUMBER OF LAYER
C
READ(IN,*)LAYER
C
C  READ IN THE NUMBER OF NODE IN EACH LAYER,THE NUMBER OF NODE IN
C  INPUT LAYER IS IN NNODE(0).
C
READ(IN,*)(NNODE(I),I=0,LAYER)
C
C  READ IN METHOD, (0 FOR PENALTY METHOD, 1 FOR STOP TRAINING METHOD)
READ(IN,*) METHOD
C
C  READ IN SEED NUMBER, TOLERANCE, W0 AND LAMDA
C
READ(IN,*) SEED, W0, LAMDA
CLOSE (UNIT = IN)
C
RETURN
END
C*****
SUBROUTINE PRINT3D(A,MLAYR,NLAYR,MNODE,
+ NNODE)
C*****
C  THIS SUBROUTINE IS TO COPY A ORIGINAL MATRIX TO NEW MATRIX. *
C  IT IS USED TO COPY TG *
C*****
INTEGER MLAYR,NLAYR,MNODE,NNODE(0:MLAYR)
DOUBLE PRECISION A(MLAYR,MNODE,0:MNODE)
INTEGER I,J,K,KK,LL
DO 10 K=1,NLAYR
KK=NNODE(K)
LL=NNODE(K-1)
DO 20 J=1,KK
DO 30 I=0,LL
WRITE(*, 100)K,J,I
100  FORMAT(1X,'LAYER # ',I5, 'J# ', I5, 'I# ',I5)
WRITE(*, 200)A(K,J,I)
200  FORMAT(1X,'A VALUE: ',E15.7)
30  CONTINUE
20  CONTINUE
10  CONTINUE
RETURN
END
C*****
FUNCTION DRANDOM(DL)
C*****
C  THIS FUNCTION IS TO CREATE A RANDOM NUMBER BETWEEN *

```

```

C      0 TO 1
C*****
DOUBLE PRECISION DL, DRANDOM
C
10 DL=DMOD(16807.0D0*DL,2147483647.0D0)
   DRANDOM=DL/2147483648.0D0
   IF(DRANDOM.LE.0.0D0 .OR. DRANDOM.GE.1.0D0) GO TO 10
   END
C*****
SUBROUTINE SENSITIVITY(SENSI,W,NLAYR,MLAYR,NNODE,
+ MNODE,T,OUT,N,SN,MSAMP)
C*****
C      THIS SUBROUTINE IS TO CALCULATE THE
C      SENSITIVITY DEFINED IN (3.6.6). PLEASE REFER
C      TO (3.6.6)-(3.6.16)
C      SENSI(MLAYR,MNODE)--THE SENSITIVITY MATRIX. REF(3.6.12)
C      W(MLAYR,MNODE,0:MNODE)--WEIGHT MATRIX
C      T(MSAMP,MNODE)--THE DESIRED OUTPUT OF THE NET
C      OUT(MSAMP,MNODE)--THE CALCULATED OUTPUT OF THE NET
C      N(MLAYR,MNODE)--THE SUMMATION OF THE WEIGHT REF(3.1.1)
C      SN -- THE SAMPLE INDEX.
C*****
INTEGER NLAYR,MLAYR,NNODE(0:MLAYR),I,J,K,KK,LL,
+ MNODE,MSAMP,SN
DOUBLE PRECISION W(MLAYR,MNODE,0:MNODE),
+ SENSI(MLAYR,MNODE),T(MSAMP,MNODE),
+ OUT(MSAMP,MNODE),N(MLAYR,MNODE),SUM
C
C      CALCULATE THE SENSITIVITY OF FINAL LAYER (3.6.16)
C
C      KK=NNODE(NLAYR)
C      DO 10 I=1,KK
C      SENSI(NLAYR,I) = -(T(SN,I) - OUT(SN,I))
+      * SIGFD(N(NLAYR,I))
10 CONTINUE
C
C      CALCULATE THE SENSITIVITY OF EACH LAYER STARTING
C      FROM THE FINAL LAYER. (3.6.12)
C
C      LOOP OVER LAYER
C
C      DO 20 K=NLAYR-1,1,-1
C      KK=NNODE(K)
C      LL=NNODE(K+1)
C      DO 40 I = 1,KK
C      SUM =0.D0
C      DO 30 J=1,LL
C      SUM = SUM + SIGFD(N(K,I))*W(K+1,J,I)*SENSI(K+1,J)
30 CONTINUE
C      SENSI(K,I) = SUM
40 CONTINUE
20 CONTINUE
C
C      RETURN

```

```

      END
C*****
      FUNCTION SIGF(X)
C*****
C      SIGMOID TRANSFER FUNCTION      *
C      INPUT: DOUBLE PRECISION: X    *
C      OUTPUT: DOUBLE PRECISION: SIGF *
C*****
      DOUBLE PRECISION X, SIGF
      SIGF = 1.D0 / (1.D0 + EXP(-X))
      RETURN
      END
C*****
      FUNCTION SIGFD(X)
C*****
C      DERIVATIVE OF SIGMOID FUNCTION *
C      INPUT: DOUBLE PRECISION: X    *
C      OUTPUT: DOUBLE PRECISION SIGFD *
C*****
      DOUBLE PRECISION X, SIGFD
      SIGFD = EXP(-X) / ((1.D0 + EXP(-X))**2)
      RETURN
      END
C*****
      SUBROUTINE SUMGRAD(G,NLAYR,MLAYR,NNODE,
+ MNODE,TG)
C*****
C      THIS FUNCTION IS TO SUM UP THE GRADIENTS      *
C      OF EACH EPOCH.                                *
C      TG(MLAYR,MNODE,0:MNODE) - STORES            *
C      THE TOTAL GRADIENTS OF NUMOFSAMPLE SAMPLES. *
C      REF. ALGORITHM 3.6.1 (2.2)                   *
C      G(MLAYR,MNODE,0:MNODE)--THE GRADIENT OF THE NET OF *
C      ONE SAMPLE.                                  *
C      TG(MLAYR,MNODE,0:MNODE)-- THE TOTAL(SUMMATION) GRADIENT*
C      OF ALL SAMPLES. REF. ALG(2.3)                *
C*****
      INTEGER NLAYR, MLAYR,MNODE,I,J,K,KK,LL,
+ NNODE(0:MLAYR)
      DOUBLE PRECISION G(MLAYR,MNODE,0:MNODE),
+ TG(MLAYR,MNODE,0:MNODE)
C
C      CALCULATE THE GRADIENT OF PERFORMACE FUNCTION W.R.T
C      WEIGHTS ACCORDING TO (3.6.6)
C      LOOP OVER LAYER
C
      DO 10 K=1,NLAYR
      KK=NNODE(K)
      LL=NNODE(K-1)
      DO 20 J=1,KK
      DO 30 I=0,LL
      TG(K,J,I)=TG(K,J,I) + G(K,J,I)
30      CONTINUE
20      CONTINUE

```

```

10    CONTINUE
      RETURN
      END
C*****
      SUBROUTINE SUMWEIGHT(P,O,N,MLAYR,NLAYR,MNODE,A,
+   NNODE,W,SN,MSAMP)
C*****
C     THIS SUBROUTINE IS TO CALCULATE THE SUM OF          *
C     THE INPUTS OF A NEURON J IN LAYER K                *
C     PLEASE REFER TO (3.1.1)                             *
C     N(MLAYR,MNODE)--STORES THE SUM OF INPUTS OF        *
C     NEURON J IN LAYER K                                *
C     A(0:MLAYR,MNODE)--STORES THE OUTPUT OF             *
C     NEURON J IN LAYER K                                *
C     A(0:MLAYR,0:MNODE) -- STORES THE INPUT DATA.      *
C     P(MSAMP,MNODE) -- IS THE INPUT DATA FROM ONE     *
C     SAMPLE                                              *
C     O(MSAMP,MNODE) -- IS THE DESIRED OUTPUT DATA FROM *
C     ONE SAMPLE                                          *
C     Q(MSAMP,MNODE) -- IS THE OUTPUT CALCULATED FROM THE *
C     NET.                                                *
C     W(MLAYR,MNODE,0:MNODE) -- THE WEIGHT OF THE NET.   *
C     SN -- THE SAMPLE INDEX.                             *
C*****
      INTEGER MLAYR,MNODE,NNODE(0:MLAYR),I,J,K,
+   NLAYR,L,SN,MSAMP
      DOUBLE PRECISION N(MLAYR,MNODE),A(0:MLAYR,0:MNODE),
+   W(MLAYR,MNODE,0:MNODE),SUM,P(MSAMP,MNODE),
+   O(MSAMP,MNODE)

C     STORE INPUT DATA INTO A(0,MNODE)

      DO 100 I=1, NNODE(0)
        A(0,I)=P(SN,I)
C     PRINT *, 'SN= ',SN,'P(SN,I)= ',P(SN,I)
100    CONTINUE

C STORE THE BIAS
      A(0,0) = 1.D0

C
C     CALCULATE THE SUM OF THE INPUTS OF A NEURON J IN LAYER K
C
C     LOOP OVER LAYER
C LOOP OVER LAYER
      DO 10 K=1,NLAYR
C LOOP OVER CURRENT NODE (TARGET)
      DO 20 J=1,NNODE(K)
C LOOP OVER PREVIOUS NODE (SOURCE)
      SUM = 0.0D0
      DO 30 I=0,NNODE(K-1)
        SUM = SUM + W(K,J,I)*A(K-1,I)
30    CONTINUE
C CALCULATE THE SUM OF 1 NEURON J IN LAYER K
      N(K,J) = SUM

```

```

C      WRITE (*, 500) K,J,N(K,J)
C500  FORMAT(1X,'LAYER #',I3,' NODE #',I3,' N = ', F16.10)
C CALCULATE THE OUTPUT OF NEURON J IN LAYER K
      A(K,J) = SIGF(N(K,J))
C      WRITE (*, 400) K,J,A(K,J)
C400  FORMAT(1X,'LAYER #',I3,' NODE #',I3,' A = ', F16.10)
20    CONTINUE
C THE BIAS
      A(K,0) = 1.D0
10    CONTINUE

C      STORED THE OUTPUT IN A(NNODE(NLAYR))
      DO 200 I=1, NNODE(NLAYR)
      O(SN,I)=A(NLAYR,I)
200   CONTINUE
      RETURN
      END

```

VITA

Ping Jiang

Candidate for the Degree of

Master of Science

Thesis: A PENALTY METHOD TO REDUCE OVERFITTING IN ARTIFICIAL
NEURAL NETWORKS

Major Field: Computer Science

Biographical:

Personal Data: Born in Shanghai, P. R. China, July 1962
the son of Su Zen Zhu and Yun Jiang.

Education: Graduated From Shanghai #2 High School, Shanghai, P. R. China;
received Bachelor of Science Degree in Structural Engineering from Tongji
University in July 1984; received Master of Science Degree in Civil
Engineering from Oklahoma State University in December 1994;
completed requirements for the Master of Science degree at Oklahoma
State University in July 1996.

Professional Experience: Engineer, Shanghai Municipal Engineering Institute,
Shanghai, P. R. China from 1984 through 1991.