A COMPARISON STUDY OF FEEDFORWARD FULLY-CONNECTED NEURAL NETWORKS VS. CASCADE CORRELATION NETWORKS FOR PREDICTION OF SOIL MOISTURE CONTENT

By

XIAOJI LIU
Bachelor of Science
East China Institute of Technology
Nanjing, China
1982

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
December, 1994

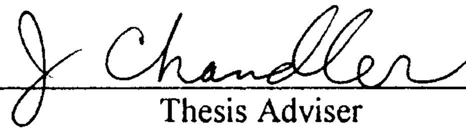
A COMPARISON STUDY OF FEEDFORWARD FULLY-CONNECTED

NEURAL NETWORKS VS. CASCADE CORRELATION

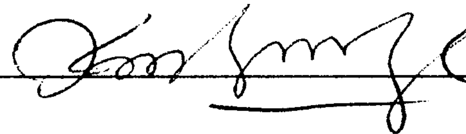
NETWORKS FOR PREDICTION OF SOIL

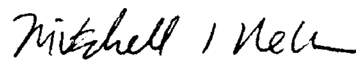
MOISTURE CONTENT

Thesis Approved:



Thesis Adviser





Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Dr. John P. Chandler, my major adviser, for his guidance, dedication, patience, and invaluable instructions. He made a great deal of effort to improve this thesis, both in content and in English. My appreciation is also extended to Dr. K. M. George and Dr. M. Neilsen, the other committee members, for their helpful advisement and suggestions.

I would also like to thank to Dr. Marvin L. Stone for providing me the opportunity to do research work on the neural networks on soil moisture prediction and permitting me to use the related experimental data in my thesis, and helping me to understand them.

I am grateful to my parents Huilin Wang and Junhua Liu, for their consistent support and encouragement. My deep thanks are also dedicated to my wife, my sisters and brothers, for their support and understanding.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 4

Basic Concepts of Neural Networks 4

Feedforward Fully-Connected Neural Networks 8

Cascade Correlation Networks 9

Description of Soil Moisture Content Prediction 14

III. LEARNING ALGORITHMS FOR NEURAL NETWORKS 17

Back-Propagation 17

Delta-Bar-Delta 23

Extended Delta-Bar-Delta 27

Quick Back-Propagation 29

Steepest Descent in Line Search 30

IV. RESULTS AND ANALYSIS 33

Test Data Preparation 33

Convergence Criterion 34

Results 36

V. CONCLUSIONS 57

A SELECTED BIBLIOGRAPHY 58

APPENDIX--PROGRAM LISTING 61

LIST OF TABLES

Table Page

I. Convergence Behavior for a Cascade Network 40

II. Convergence Behavior for Standard Back-Propagation with Three Hidden Units 41

III. Convergence Behavior for Standard Back-Propagation with Four Hidden Units 42

IV. Convergence Behavior for Standard Back-Propagation with Five Hidden Units 43

V. Convergence Behavior for Quick Back-Propagation with Three Hidden Units 44

VI. Convergence Behavior for Quick Back-Propagation with Four Hidden Units 45

VII. Convergence Behavior for Quick Back-Propagation with Five Hidden Units 46

VIII. Convergence Behavior for Steepest Descent with Three Hidden Units 47

IX. Convergence Behavior for Steepest Descent with Four Hidden Units 48

X. Convergence Behavior for Steepest Descent

with Five Hidden Units 49

XI. Convergence Behavior DBD with Three Hidden Units 50

XII. Convergence Behavior DBD with Four Hidden Units 51

XIII. Convergence Behavior DBD with Five Hidden Units 52

XIV. Convergence Behavior EDBD with Three Hidden Units 53

XV. Convergence Behavior EDBD with Four Hidden Units 54

XVI. Convergence Behavior EDBD with Five Hidden Units 55

XVII. Comparison of Convergence Behavior for Different Architectures

and Algorithms 56

LIST OF FIGURES

Figure Page

1. Input Summation and Output Modification 5

2. Sigmoid Function 5

3. Threshold Function 6

4. Hypertangent Function 6

5. General Feedforward Fully-Connected Network 9

6. Cascade Architecture: Initial State with No Hidden Units 10

7. Cascade Architecture with One Hidden Unit 11

8. Cascade Architecture with Two Hidden Units 11

9. Cascade Architecture with Three Hidden Units 12

10. Feedforward Fully-Connected Network with
One Hidden Layer with Five Units 16

11. $\alpha(k)$--Linear Function 25

12. $\alpha(k)$--Exponential Function 26

CHAPTER I

INTRODUCTION

Artificial neural networks have many applications in solving problems of prediction such as stock prices, grain harvest [4], etc. [3] described a feedforward fully connected neural network with learning algorithm of standard back-propagation that can predict the soil moisture content, and gave good results. However, the work on comparison of performance of this neural network for prediction of soil moisture with several other alternatives has not been done. This comparison would involve choice of different training algorithms with the same neural network architecture, and the choice of different network architectures. This is obviously of great interest to us.

For training layered feedforward neural networks, back-propagation is the most frequently applied algorithm [5]. However, the standard back-propagation has the problem of choosing a step size [7] since it just computes $\frac{\partial E}{\partial w_i}$, the partial first derivative of the overall error function E with respect to each weight w_i in the network. When these derivatives are given, a gradient descent can be performed in the weight space, reducing the error with each step. Clearly, if we take infinitesimal steps down the gradient vector, running a new training epoch to recompute the gradient after each step, we will eventually reach a local minimum of the error function. Experience has shown that in most cases, this local minimum will be a global minimum, or at least a good enough solution of the problem. But actually we can't take infinitesimal steps from a practical point of view; instead we always want to take steps that are as large as

possible so that we can speed up the learning process. Unfortunately, if we choose a step size that is too large, the networks may not converge to the solution we desire.

Many schemes have been suggested to deal with the step size problem. Fahlman's quick propagation is one of them. Quick propagation not only considers the first partial derivative $\frac{\partial E}{\partial w_i}$ but also uses a second order method that is related to Newton's method, to update the weights.

Another scheme to deal with step size involves dynamically adjusting the step size of learning, based on the change in gradient between successive steps [1], [2], [4]. In this thesis, this kind of method is called Delta Bar Delta (DBD) as in [4]. Modification to the method of Delta Bar Delta (DBD) will lead to the method of Extended Delta Bar Delta (EDBD) [4].

Minimization techniques have also been explored to solve the step size problem. Conjugate gradient method with line search and scaled conjugate gradient method without line searches have been studied for this purpose [10], [15]. But in this thesis, we will study gradient descent with a line search.

One of the problems with feedforward fully connected neural networks is that the architecture has to be specified beforehand; i.e., the number of hidden layers as well as the number of neuron units in each layer must be determined. But, most of time it is difficult to know how many hidden layers and how many neuron units in each layer are appropriate to solve particular applications. Fahlman's Cascade Correlation network [7] provides an approach to deal with this problem. A Cascade Correlation network just requires a fixed number of neuron units in the input layer

and output layer, which are actually application dependent, before training begins. It just adds one unit each time in the hidden layer during the training course. Therefore, it not only speeds up learning, but also saves storage for weights and neurons and helps avoid overfitting the data. In addition, according to Fahlman [7], it can solve the problem of a moving target.

This thesis is organized as follows:

In Chapter I, a general introduction to the problem we are going to investigate is given.

In Chapter II, a brief review will be given of neural network basic concepts, feedforward fully connected networks, the cascade correlation network, and a description of the soil moisture content prediction problem.

Chapter III will be dedicated to the study of five training algorithms, which are standard back propagation, quick propagation, delta bar delta (DBD), extended delta bar delta (EDBD), and steepest descent in batch mode with line search.

In Chapter IV, we will give the results of training and testing neural networks for prediction of soil moisture content using two different architectures and five different training algorithms.

In Chapter V, we will make some conclusions on the comparison of performance of these two neural networks as well as five different training algorithms for prediction of soil moisture.

Finally, the source program which implemented standard back-propagation, quick back-propagation, delta bar delta, extended delta bar delta, and minimization with line search will be put into Appendix A.

Chapter II

Literature Review

Basic Concepts of Neural Networks

The neuron is the fundamental cellular unit of the nervous system and the brain. Each neuron is a simple microprocessing unit which receives and combines signals from many other neurons through input processes. If the combined signal is strong enough it activates the firing of the neuron which produces an output signal. In artificial neural networks, the unit analogous to the biological neuron is referred to as a processing element (PE). A processing element has many input paths and combines them by a simple summation of the values of these inputs. This can be described as follow:

$$I_i = \sum_j w_{ij} x_j$$

The combined input is then modified by a transfer function or "squashing" function. There are various forms of transfer function, which can be a threshold function that only passes information if the combined activity level reaches a certain level, or it can be a continuous function such as a sigmoid function or hypertangent function. The output function can be represented as follows:

$$O_i = f(I_i)$$

The input summation and output modification is shown in Fig 2.1

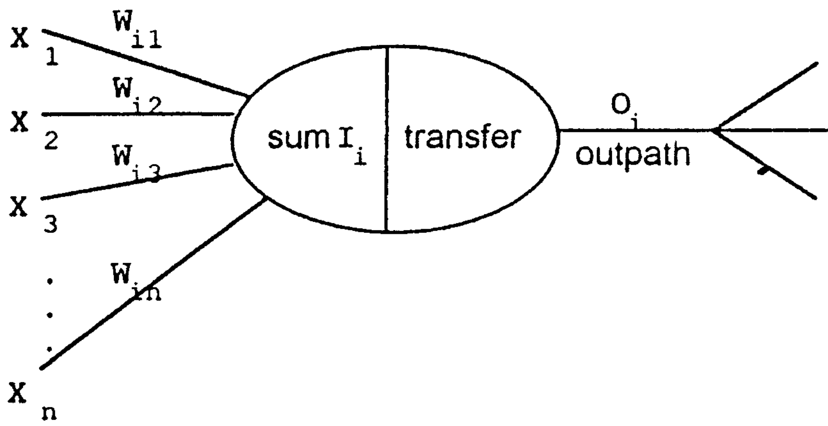


Figure 2. 1

There are several functions that can used as transfer functions, which can be described as follow. A sigmoid function is defined as below

$$f_1(z) = \frac{1}{1+e^{-z}}$$

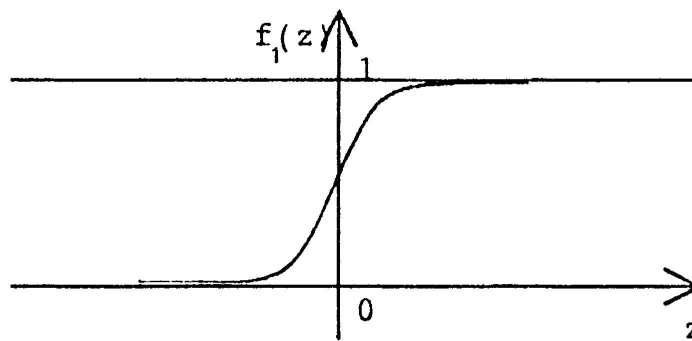


Figure 2. 2

The threshold function is defined as

$f_2(z) = 1$ if $z > T$

$f_2(z) = 0$ otherwise

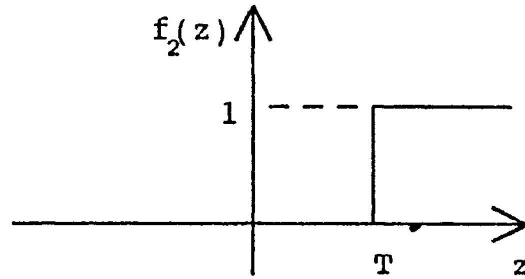


Figure 2. 3

The hypertangent function is defined as

$$f_3(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

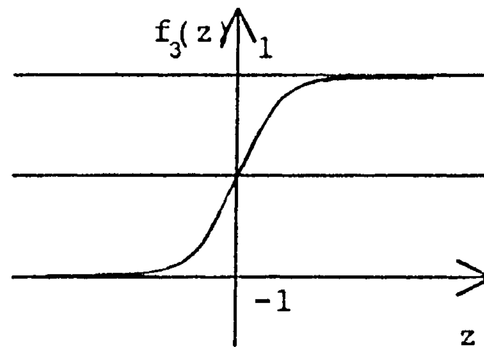


Figure 2.4

Note that $f_3(z)$ is related to $f_1(z)$ by $f_3(z) = 2f_1(2z) - 1$.

The output path of a processing element can be connected to the input paths of other processing elements through connection weights. A neural network consists of many such processing elements together and very interesting effects result from the ways the neurons are interconnected.

Processing elements are usually organized into groups called layers. Generally there are two layers that provide a connection from networks to the outside world: an input layer where data is presented to the network and an output layer which holds the response of the network to a given input. The layers between the input layer and the output layer are called hidden layers.

There are two phases in the iteration of a neural network, learning and recall [4]. Learning is the process of adapting or modifying the connection weights in response to input vectors presented to the input layer. If there is a desired output presented at the output layer, we call this supervised learning. There are many learning algorithms existing for such a learning process. There are Hebbian learning, the Delta rule, etc.. The most popular one may be back-propagation, which we will discuss in Chapter III in detail.

One of the important properties of a neural network is its capability of storing information. Neural computing is distributed and the connection weights are the memory units of a neural network. The nature of a neural network memory leads to a reasonable response when the network is presented with a previously unseen input. This property is referred to as generalization. The quality of generalization depends on the particular application and on the sophistication of the network. Feedforward fully connected networks with back-propagation learn about the features in their hidden layers. The knowledge in the hidden layers can be combined to form intelligent responses to novel stimuli [4], [2]. Some efforts were made to improve the generalization performance of neural networks. [11] proposed a scheme called double propagation to get better generalization from a training set to a test set. The idea of this method is to form an energy

function that is the sum of the normal energy term found in general back-propagation and an additional term that is a function of the Jacobian. [14] showed that the improvements are especially significant for those architectures that show good performance when trained using back-propagation.

Feedforward Fully Connected Neural Networks

The simplest form of a network has no feedback connection from one layer to another or to itself. Such a network is called a feedforward network. In a feedforward network, information is passed from the input layer through the hidden layers to the output layer, in each of which a summation and a transfer function are used. Furthermore, if each unit in one layer in the network is just connected to the layer immediately below it or above it, we call it a feedforward layered network or feedforward fully connected network. Clearly, in feedforward networks, each layer can only receive signals from the immediately previous layer and send signals to the immediately following layer. A feedforward fully connected network is shown in Fig 2.5

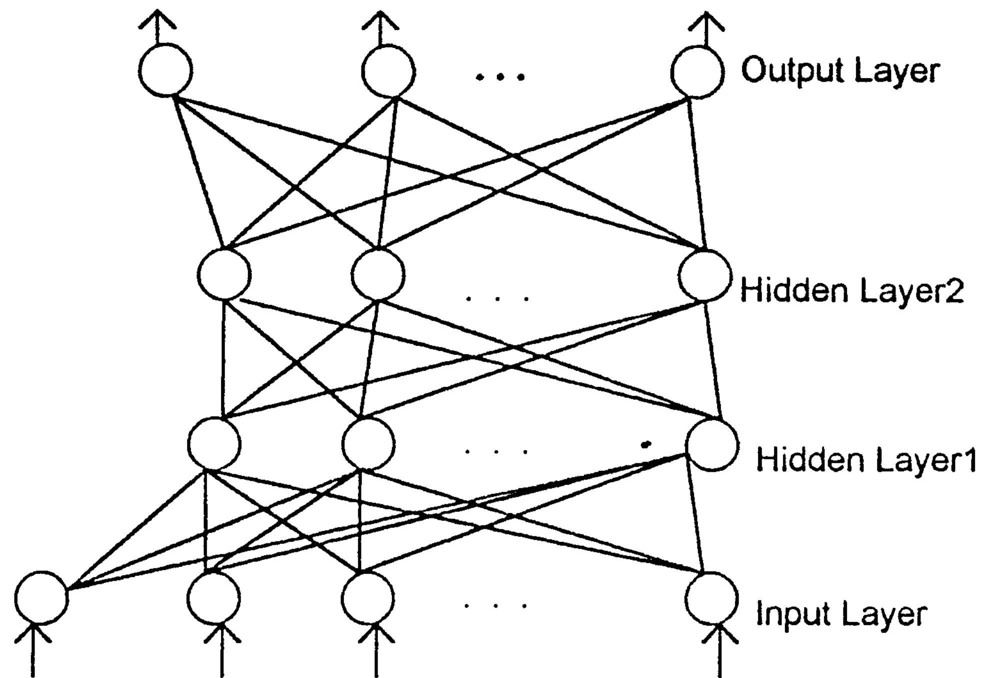


Figure 2.5

Cascade Correlation Networks

The Cascade correlation network was proposed by S. E. Fahlman to deal with the so-called moving target problem [4]. Unlike feedforward fully connected networks, a cascade correlation network does not have to be specified by a fixed number of hidden layers as well as a specified number of neuron units in each hidden layer. Instead, it just has a minimal topology at the beginning of learning, and then adds new hidden units one by one during the training course, thus creating a multilayer structure.

Fig 2.6 shows a sample cascade correlation network architecture which has six inputs, two outputs, and a bias that is permanently set to 1.0. This is a minimal structure for a cascade correlation network. Clearly, this minimal structure is application-dependent, i.e., the number of inputs and

number of outputs are determined by the particular application. All of the inputs and the bias directly connect to the outputs.

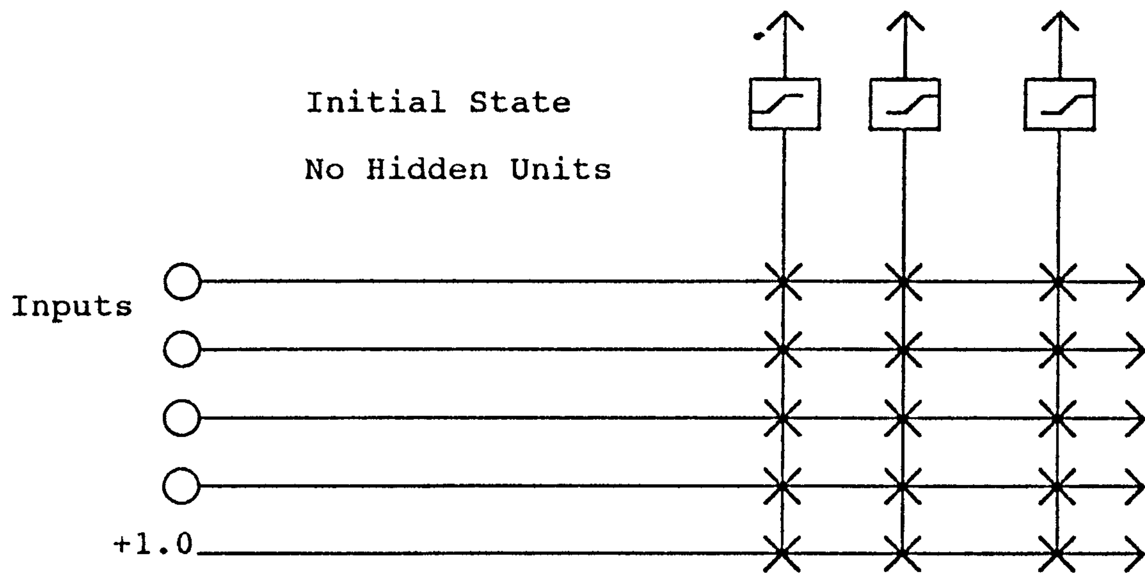


Figure 2. 6 The Cascade architecture: Initial state with no hidden units

15 weights, one at each X

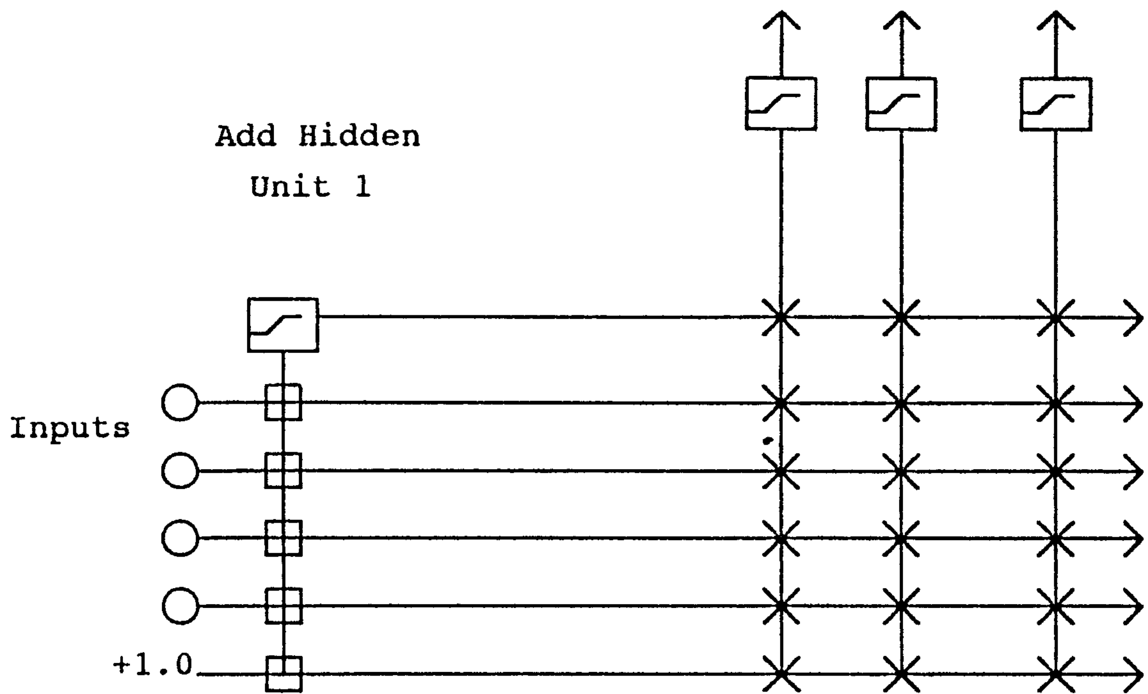


Figure 2.7 Cascade architecture with one hidden unit

18 weights, one at each X

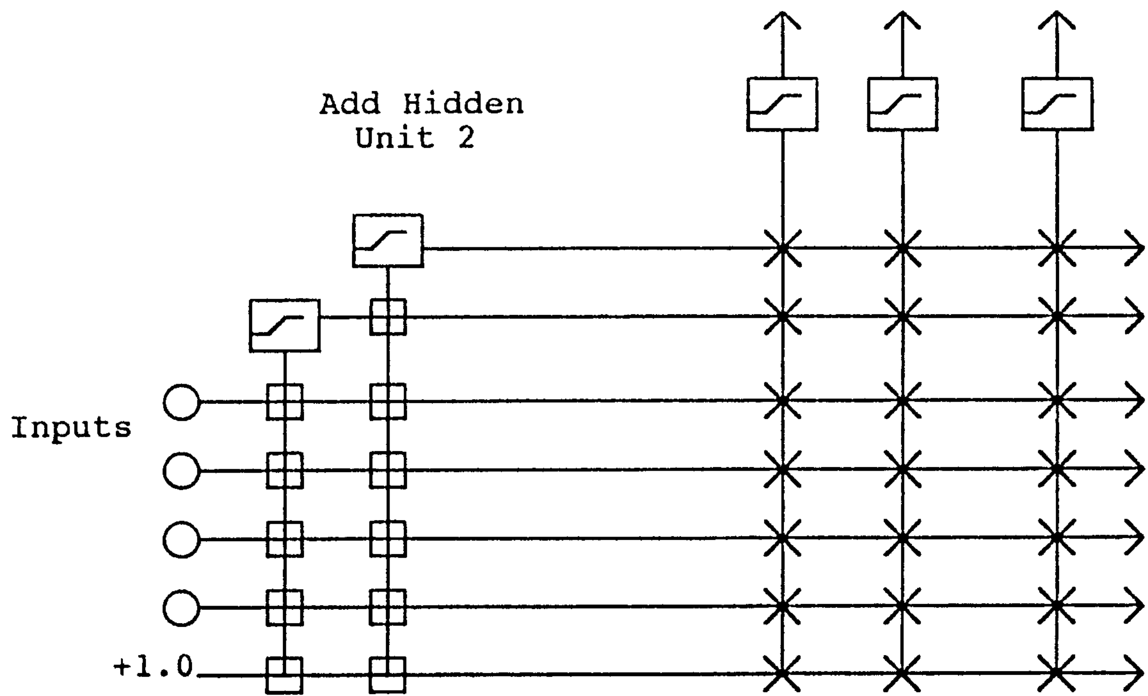


Figure 2.8 Cascade architecture with two hidden units

21 weights, one at each X

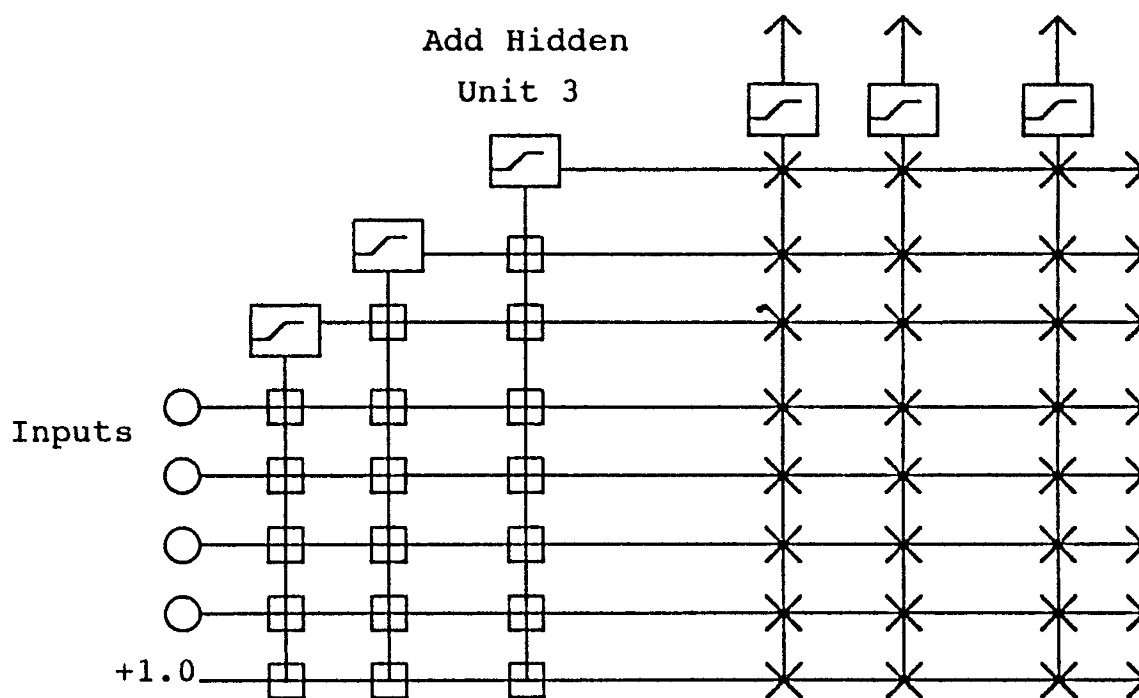


Figure 2.9 Cascade architecture with three hidden units

21 weights, one at each X

where the vertical lines sum all incoming activations. Boxed connections are frozen and X connections are trained repeatedly.

At first, the training begins with no hidden units. The connection weights between inputs and outputs are directly trained as well as possible over the training set. This process can be repeated until some criterion is satisfied. In Fahlman's implementation, there are two parameters governing this process, which we will discuss in detail in Chapter V. Since this is just a single layer network, several learning algorithms can be chosen for training, which include Widrow-Hoff or the Delta rule, the perceptron learning algorithm, etc. In Fahlhman's implementation, quick propagation was chosen as the learning algorithm.

After a number of epochs of training the network, which is set in a parameter, if the accuracy is still not satisfied, a hidden unit is added to the existing network. This new hidden unit will receive a connection from each of the network's original inputs and also from each of the pre-existing trained hidden units. The input connection weights of this new hidden unit can be decided as below.

We begin with a candidate unit that receives input connections as indicated above. To adjust these connection weights, we introduce a correlation function S, which is defined below [4]:

$$S = \sum_{o} \sum_{p} (v_p - \bar{v})(E_{p,o} - \bar{E}_o)$$

where o is the network output at which the error is measured and p are the training examples or patterns. The $\bar{V}$ and $\bar{E}_o$ are values of V and E_0 averaged over all training examples. V is the candidate unit's value, and E_0 is the residual output error observed at unit O. The goal is to maximize the function S. In order to do this, we need to calculate the partial derivative of S with respect to each of the candidate unit connection weights, $\frac{\partial S}{\partial W_i}$.

This can be represented as

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} s_o (E_{p,o} - \bar{E}_o) f'_p I_{i,p}$$

where σ_o is the sign of the correlation between the candidate value and the output O, f'_p is the derivative for training example p of the candidate unit's activation function with respect to the sum of its inputs, and $I_{i,p}$ is the input that the candidate unit receives from unit i for example p.

After computing $\frac{\partial S}{\partial w_i}$ for each incoming connection, we can perform a gradient ascent to maximize S. So we can adjust the input connection weights by using an appropriate learning algorithm, for example, quick back-propagation. When S stops improving, we can add this new candidate as a new unit to the network.

Instead of using a single candidate, [4] uses a pool of candidate units, where each candidate unit is set to a different random initial weight and receives the same input signals, and sees the same residual error for each training pattern. These candidates can be trained separately or in parallel, so they will receive different input connection weights. When this training stops, we can pick the one from the pool whose correlation score is the best. The advantage of using a pool of candidates is that it can greatly reduce the chance that a useless unit will be permanently installed since an individual candidate unit may get stuck during training. In [4], the size of the pool is chosen to be 12.

When the candidate has been created, it can be installed in the existing network. The candidate's input connection weights will be frozen, while its output connection will be trained repeatedly until the error satisfies the convergence criterion.

Description of Soil Moisture Content Prediction

The soil moisture content measure is important in agricultural engineering. It varies with depth, time, texture, bulk density, climate and many other factors [3]. However, it is difficult to get an instantaneous,

accurate measure of soil moisture. Since the rate of heat dissipation is sensitive to water content according to soil thermal theory, we can predict soil moisture by using soil temperatures, and soil temperature is much easier to measure than soil moisture.

[3] indicates that the soil moisture at some depth from the soil surface is related to the soil temperatures at different levels of depth. Also the soil moisture at time t correlate to the temperature at time t-k, where k is a time constant. Generally k is set to 12 hours [3]. This means that the moisture relates to the temperature 12 hours before. For example, the soil moisture of a depth of 30 cm is correlated with the temperatures at depths of 10, 20, 40, 50 cm respectively. Furthermore, for the same level of depth, three sample site data are used. Now we can decide how many input units are required in the network for this application. We have four levels of depth of temperatures, each level with three sampling sites. So for time t, we have 4*3 = 12 data entries. In addition, since we need this sort of data 12 hours before, we have another 12 data entries. So a total of 24 temperature data entries are required. Also, we always have a bias that is permanently set to 1.0. For the output layer, we need only one unit as moisture output. For choosing the number of hidden layers, [4] indicate that one or two hidden layers are enough for most applications. For choosing the number of neuron units in one hidden layer, we will try several different numbers to get best performance of the network. Figure 2.10 shows a feedforward fully connected network with one hidden layer with five neuron units.

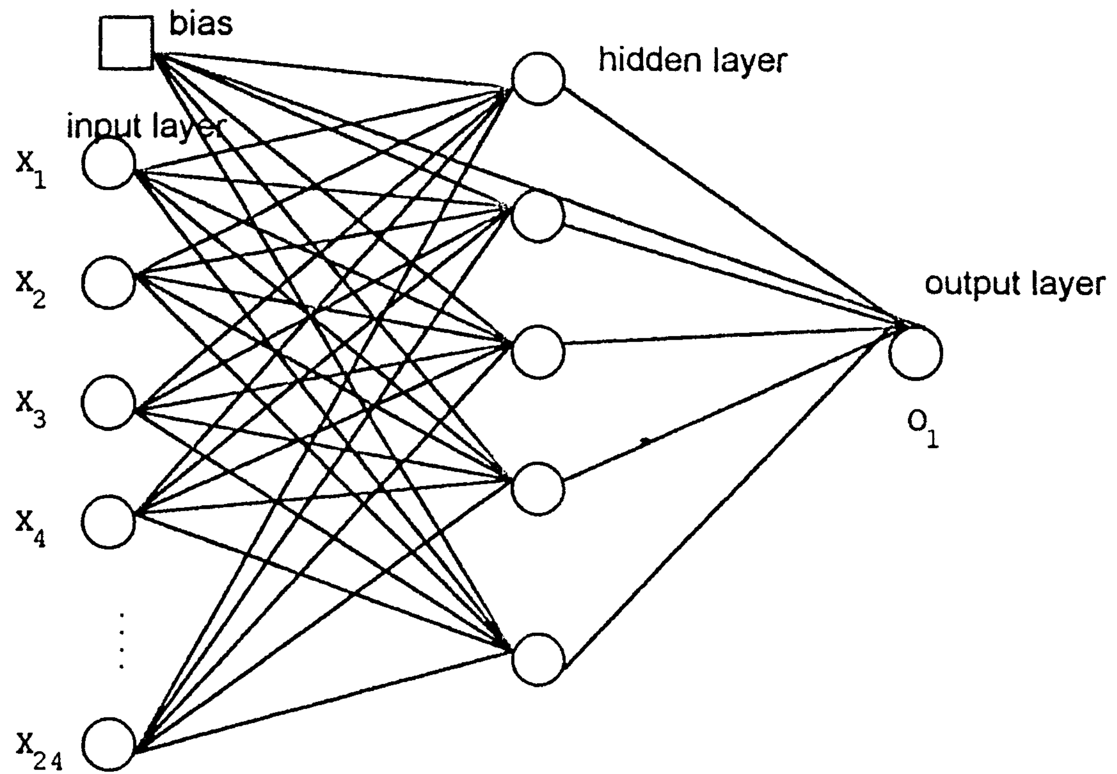


Figure 2.10

For a cascade correlation architecture, the number of input units and output units is the same as in a fully connected network, but the number of hidden layers as well as the number of units in each layer is dynamically determined during the training course. We just need to assume a minimal structure for a cascade correlation architecture at the beginning of training, i.e. the input layer and output layer.

Chapter III

Learning Algorithms for Neural Networks

Back-Propagation

The back-propagation method of Rumelhart, Hinton, and Williams [12] is a learning procedure for multilayer feedforward neural networks. By means of this procedure, the network can learn to map a set of inputs to a set of outputs. The mapping is specified by giving the desired activation state of the output units for each presented state of the input units. Learning is then carried out by iteratively adjusting connection weights in the network so as to minimize the differences between the actual output state vector of the network and the desired output state vector. During the learning process, an input vector is presented to the network and propagated forward to determine the output signal. The output vector is compared with the desired output vector, thus resulting in an error signal, which is back-propagated through the network in order to adjust the connection weights in the network. This procedure will be repeated until the network converges to a state that is sufficiently close to the desired one. Back-propagation can be described as below.

Here we consider a network with N input neurons (processing elements), M outputs and an arbitrary number of hidden layers. We assume that each neuron output is fully connected to the immediately following layer; i.e., from input to output.

The typical back-propagation network always has an input layer, an output layer and at least one hidden layer. There is no theoretical limit on the number of hidden layers but typically there are one or two. [2] indicate that maximum of four layers (three hidden layers and one output layer) are required to solve arbitrarily complex pattern classification problems. Each layer is fully connected to the succeeding layer.

For convenience, we define notation as follows:

$X = (x_1, x_2, x_3, \dots x_m)$	input vector
$Y = (y_1, y_2, y_3, \dots, y_m)$	desired output vector
$S = (s_1, s_2, s_3, \dots, s_m)$	actual obtained output vector
S^k	actual obtained output vector at k'th iteration
Y^k	desired output vector at k'th iteration
s_i^k	i'th component of S at k'th iteration
y_i	i'th component of Y at k'th iteration
f	the activation function of a neuron
f'	the derivative of f
O_j	the output of neuron j
I_j	the input of neuron i
$e(k)$	the step size at iteration k

The total error in the output when one training example is presented to the input layer is

$$E^k(w) = (S^k - Y^k)^2 = \sum_{i}^{m} (S_i^k - Y_i^k)^2$$

The total error over the complete training set is then calculated:

$$E(w) = \sum_{k} E^k(w)$$

The back-propagation algorithm consists of carrying out a gradient descent minimization process on E. In general, an approximation may be used, in which each connection weight is modified following each presentation of example k, using changes given by:

$$w_{ij}(k) = w_{ij}(k-1) - e(k)\frac{\partial E^k}{\partial w_{ij}}$$

This requires the program to calculate the sensitivity of E^k to each weight w_{ij}:

$$\frac{\partial E^k}{\partial w_{ij}} = \frac{\partial E^k}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}}$$

Alternatively:

$$\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial(\sum_p w_{ip} O_p)}{\partial w_{ij}} = O_j$$

In equation (32.3), p ranges over the neurons in the layer preceding neuron i, and the outputs O_p of these neurons do not depend on the weights w_{ij}. The following result for the error sensitivity can be obtained:

$$\frac{\partial E^k}{\partial w_{ij}} = \frac{\partial E^k}{\partial I_i} O_j$$

Substituting d_i for $\frac{\partial E^k}{\partial I_i}$, we obtain:

$$\frac{\partial E^k}{\partial w_{ij}} = d_i O_j$$

thus giving

$$w_{ij}(k) = w_{ij}(k-1) - e(k) d_i O_j$$

For neuron i in the output layer, since only s_i^k depends on I_i, we have:

$$d_i = \frac{\partial\left[\sum_j (s_j^k - y_j^k)^2\right]}{\partial I_i} = 2(s_i^k - y_i^k)\frac{\partial s_i^k}{\partial I_i}$$

Furthermore, since $s_i^k = f(I_i)$:

$$d_i = 2(s_i^k - y_i^k) f'(I_i)$$

for the neurons in the hidden layers:

$$d_i = \sum_h \frac{\partial E^k}{\partial I_h} \frac{\partial I_h}{\partial I_i} = \sum_h d_h \frac{\partial I_h}{\partial I_i}$$

In this equation, h ranges over the neurons to which neuron i sends signals. In reality, the inputs I to other neurons are independent of I_i. This means that

$$d_i = \sum_h d_h \frac{\partial I_h}{\partial O_i} \frac{\partial O_i}{\partial I_i}$$

Using an index p over the neurons providing input to h, these neurons are contained in the same layer as i and thus their outputs O_p are independent of O_i for $p \neq i$, giving

$$\frac{\partial I_h}{\partial O_i} = \frac{\partial(\sum_p w_{hp})}{\partial O_i}$$

Finally, since $O_i = f(I_i)$, we obtain

$$d_i = \sum_h d_h w_{hi} f'(I_i)$$

This gives the complete rule for modifying the weights, when an example from the training set is presented for the k'th time:

$$w_{ij}(k) = w_{ij}(k-1) - e(k) d_i O_j$$

$$d_i = 2(s_i - y_i) f'(I_i) \qquad \text{(output layer)}$$
$$d_i = \sum_h d_h w_{hi} f'(I_i) \qquad \text{(hidden layer)}$$

The error function can be defined as

$$E = (\frac{1}{2}) \sum_k (d_k - o_k)^2$$

There are other alternative definitions of the error function, which include

$$E_3 = (\frac{1}{3})\sum_k |d_k - o_k|^3$$

and

$$E_4 = (\frac{1}{4})\sum_k (d_k - o_k)^4$$

Essentially back-propagation is a gradient descent algorithm. One of the problems of this method is that it needs to set an appropriate learning rate. Changing the connection weights as a linear function of the partial derivatives as defined above makes the assumption that the error surface is locally linear, where "locally" is defined by the size of the learning rate. However, at some point of high curvature this linearity does not hold and divergent behavior might occur at such points. It is therefore important to keep the learning coefficient low enough to avoid such behavior. But on the other hand, a small learning rate can lead to very slow learning. A momentum term was introduced to deal with this problem [4]. The weight Δw_{ij} at time t is modified so that the Δw_{ij} at time t-1 is added to it and feeds through to the current delta weights. So the delta weights can be defined as

$$\Delta w_{ij}(t) = \varepsilon e_j x_i + \eta \Delta w_{ij}(t-1)$$

where ε is the learning rate and η is the momentum coefficient.

Even though adding a momentum term, some problems may still exist with learning speed. Intuitively, different weights should have different learning rates and different momentum coefficients. So several schemes of

dynamically adjusting the learning rate and momentum coefficients have been proposed [4], which we will discuss in detail in later sections of this chapter.

Delta-Bar-Delta (DBD)

Delta-Bar-Delta is a heuristic approach to improving the rate of convergence of the connection weights in a multilayer neural network [1]. Generally speaking, each component of the weight vector may be quite different in terms of its effect on the overall error surface. In particular, every connection of a network should has its own learning rate. The step size appropriate for one component of the weight vector may not be appropriate for another weight component. Furthermore, these learning rates should vary with time. The standard feedforward networks usually have only a single learning rate for all connections, or a single learning rate for all connections in the same layer. Permitting the learning rate for each connection in the neural network to change continuously over time may speed up connection weight convergence.

Since there are a lot of connection weights in a neural network, it is very complex to determine how each weight varies over time. One scheme for adjusting the connection weights was proposed in [2]. The basic idea behind this is that, when the sign of the increment in a weight changes for several consecutive time steps, the learning rate for that connection weight should be decreased, while if the connection weight changes have the same sign for several consecutive time steps, the connection learning rate for that connection weight should be increased.

Here we define notation:

$E(k)$	value of the error at time k
$w(k)$	connection weight at time k
$\Delta w(k)$	connection delta weight at time k
$\alpha(k)$	connection learning rate at time k
$\Delta\alpha(k)$	connection delta learning rate at time k
$\delta(k)$	gradient component of the weight change at time k
$\bar{\delta}(k)$	weighted, exponential average of previous gradient components at time k
θ	convex weight factor
κ	constant learning rate
ϕ	constant decrement factor

The Delta-Bar-Delta algorithm is given as

$$w(k+1)= w(k)+\alpha(k)\delta(k)$$
$$\bar{\delta}(k)=(1-\theta)\delta(k)+\theta\delta(k-1)$$
$$\alpha(k)= \alpha(k-1)+\Delta\alpha(k)$$

$$\Delta\alpha(k)= \kappa \qquad \text{if } \bar{\delta}(k)\delta(k-1)> 0$$
$$\Delta\alpha(k)= -\varphi\alpha(k) \qquad \text{if } \bar{\delta}(k-1)\delta(k)< 0$$
$$\Delta\alpha(k)= 0 \qquad \text{otherwise}$$

To understand how the rule works, we consider two simple cases. Set the parameters $\kappa=\varphi=0.1$ and let $\alpha_0=2.0$ as an initial value. First, suppose that the gradient components of the weight change for a connection are of the same sign for five consecutive steps. At the end of

these iterations, the connection learning rate will have been incremented five times as shown below

$$\alpha_1 = \alpha_0 + 0.1$$

$$\alpha_2 = \alpha_1 + 0.1$$

$$\alpha_3 = \alpha_2 + 0.1$$

$$\alpha_4 = \alpha_3 + 0.1$$

$$\alpha_5 = \alpha_4 + 0.1 = \alpha_0 + 0.5 = 2.5$$

The change of α(k) is shown in Figure 3.1

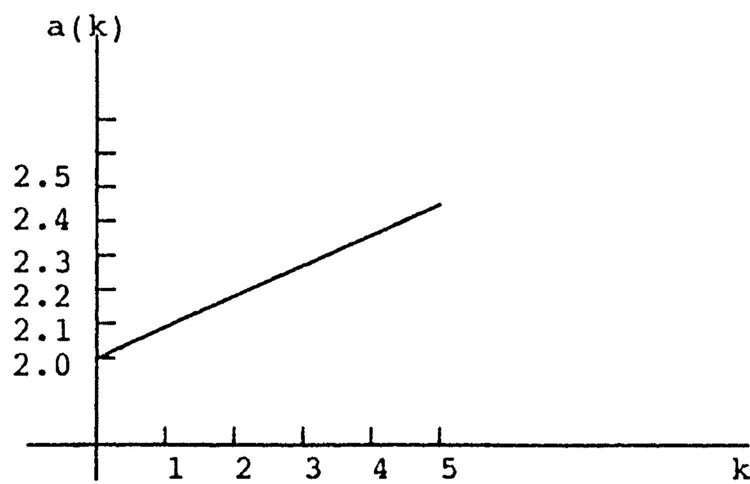


Figure 3.1

In contrast, suppose that the gradient components of the weight change for a connection alternate sign for five consecutive steps. The connection learning rate is adjusted as below:

$$\alpha_1 = \alpha_0 - 0.1\alpha_0 = (1.0 - 0.1)\alpha_0$$

$$\alpha_2 = \alpha_1 - 0.1\alpha_1 = (1.0 - 0.1)\alpha_1$$

$$\alpha_3 = \alpha_2 - 0.1\alpha_2 = (1.0 - 0.1)\alpha_2$$

$$\alpha_4 = \alpha_3 - 0.1\alpha_3 = (1.0 - 0.1)\alpha_3$$

$$\alpha_5 = \alpha_4 - 0.1\alpha_4 = (1.0 - 0.1)\alpha_4$$

$$= (1.0 - 0.1)^5\alpha_0 = 1.18098$$

The changes of $\alpha(k)$ can be shown in Figure 3.2

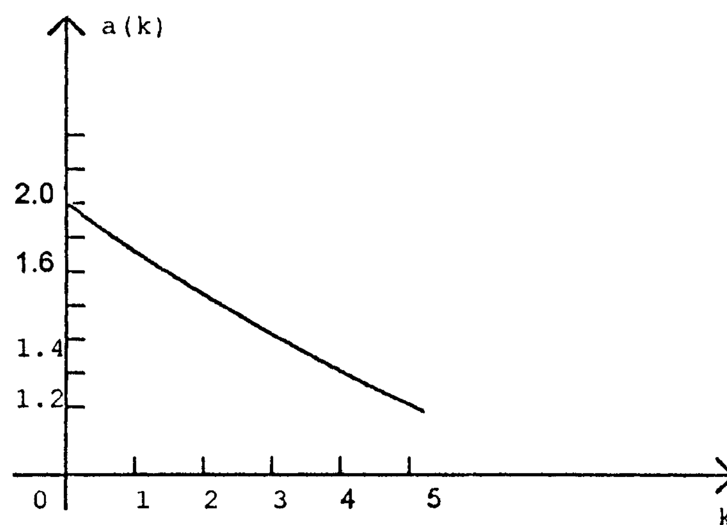


Figure 3.2

It is clear from the above cases that the rule increments learning rates linearly, but decrements them geometrically. Incrementing linearly can prevent the learning rate from becoming too large too fast. Decrementing geometrically ensures that the connection learning rates are always positive. Furthermore, they can be decreased more rapidly in regions of high curvature.

In the Delta-Bar-Delta scheme, the error calculation and propagation is the same as standard Back-Propagation. The only difference is that a varying learning rate for each connection weight is adopted.

Extended Delta-Bar-Delta (EDBD)

The extended Delta-Bar-Delta scheme was introduced to overcome some of shortcomings of Delta-Bar-Delta. Delta-Bar-Delta does not use a momentum heuristic, and even small, linear increases of k could eventually cause a learning rate to increase sufficiently that it might result in wild jumps in weight space in this scheme. Furthermore, the geometric decrease is sometimes not fast enough to prevent wild jumps.

Here we define notation as follows:

$\mu(k)$	connection momentum rate at time k
$\Delta\mu(k)$	connection delta momentum change at time k
κ_α	constant learning rate scale factor
κ_μ	constant momentum rate scale factor
γ_α	constant learning rate exponential factor
γ_μ	constant momentum rate exponential factor
φ_α	constant learning rate decrement factor
α_μ	constant momentum rate decrement factor
α_{max}	upper bound on learning rate
μ_{max}	upper bound on momentum rate

The Extended Delta-Bar-Delta scheme is given as follows:

$$\Delta w(k+1) = \alpha(k)\delta(k) + \mu(k)\Delta w(k)$$

$$\bar{\delta}(k) = (1-\theta)\delta(k) + \theta\delta(k-1)$$

$$\Delta\alpha(k) = \kappa_a e^{-\gamma_a|\delta(k)|} \qquad \text{if } \bar{\delta}(k-1)\delta(k) > 0$$

$$\Delta\alpha(k) = -\varphi_a \alpha(k) \qquad \text{if } \bar{\delta}(k-1)\delta(k) < 0$$

$$\Delta\alpha = 0 \qquad \text{otherwise}$$

$$\Delta\mu = \kappa_u e^{-\gamma_\mu|\overline{\delta(k)}|} \qquad \text{if } \bar{\delta}(k-1)\delta(k) > 0$$

$$\Delta\mu = -\varphi_\mu \mu(k) \qquad \text{if } \bar{\delta}(k-1)\delta(k) > 0$$

$$\Delta\mu = 0 \qquad \text{otherwise}$$

$$\alpha(k) = \alpha(k-1) + \Delta\alpha(k)$$
$$\mu(k) = \mu(k-1) + \Delta\mu(k)$$

To prevent wild jumps in weight space, constraints will be imposed on $\alpha(k), \mu(k)$:

$$\alpha(k) \le \alpha_{max}$$
$$\mu(k) \le \mu_{max}$$

Notice that the learning rate and the momentum rate have separate constants controlling their increase and decrease. Once again, the sign of $\delta(k)$ is used to indicate whether, heuristically, an increase or decrease is appropriate. The adjustment for decrease is identical in form to that for DBD. However, the learning rate and momentum rate increases were modified to be exponentially decreasing functions of the magnitude of the weighted gradient components, $\delta(k)$. Thus, greater increases will be

applied in areas of small slope or curvature than in areas of high curvature. This is a partial solution to the jump problem.

Quick Back-Propagation

To deal with the problem of slowness of back-propagation, many schemes have been proposed. One of them is quick back-propagation, or QuickProp, proposed by Fahlman [5]. Quick back-propagation is a second-order method, based loosely on Newton's method. Two assumptions are made with this method: first that the error vs. weight curve for each weight can be approximated by a parabola whose arms open upward; second that the change in the slope of the error curve as seen by each weight is not affected by all of the other weights that change at the same time [5]. Based on these two assumptions, the delta weight Δw_{ij} can be computed as below

$$\Delta w_{ij}(t) = \varepsilon * \frac{\frac{\partial E}{\partial w_{ij}}(t)}{\frac{\partial E}{\partial w_{ij}}(t-1) - \frac{\partial E}{\partial w_{ij}}(t)} * \Delta w(t-1)$$

where ε is a learning rate and needs to be predetermined.

In this computation, we involve not only the current slope but also the previous slope in the weight space. One situation may happen when the current gradient is in the same direction as the previous gradient but is the same size or larger in magnitude. In this case we would take an infinite step or actually move backwards, up the current slope and toward a local maximum. One of parameters called μ was introduced to deal with this

problem. We will not allow a weight step that is greater than μ times the previous step for that weight. If the step computed by quickprop would be too large, infinite or uphill on the current slope, we use μ times the previous step as the size of the new step. The choice of μ depends on the application. [5] suggested that $\mu = 1.75$ will work for a wide range of problems.

Steepest Descent with line search

Since standard back-propagation has a poor convergence rate and depends on parameters which have to be specified by the user, there have been efforts to improve the performance of back-propagation. One of them is to try some minimization techniques to deal with this problem.

From an optimization point of view, learning with back-propagation in a neural network is equivalent to minimizing a global error function, which is a multivariable function that depends on the connection weights in the network. Johansson, Dowla, and Goodman [15] describe the theory of general conjugate gradient methods and how to apply the methods in feedforward neural networks. They pointed out that the standard conjugate gradient method with line search is faster than standard back-propagation when tested on the parity problems [15]. Martin introduced a new variation of the conjugate gradient method -- scaled conjugate gradient, which avoids the line search per learning iteration by using the Levenberg-Marquardt approach [15]. In this thesis, we will just investigate the gradient descent minimization with line search for training a neural network with back-propagation.

We can regard a feedforward neural network as a function

$$F = F(x_1,x_2,x_3,\ldots,x_n)$$

to be minimized where $X=(x_1,x_2,x_3,\ldots,x_n)$ are the connection weights in the network. As a matter of fact, F is the error function, and our goal is to minimize it. For a gradient descent method, the minimization search direction can be obtained from the gradient vector. The line search need to be used to find the minimum point along the search direction. So given a fixed search direction d and an initial point X, the line search problem is that we just need to find α, such that

$$F(\alpha) = F(X + \alpha d)$$

is minimized. There are several line search methods available up to now. They generally involve function evaluations and/or both function evaluation and gradient calculations. [16] studied the Brent line search method and the Nash line search method. For simplicity, in this thesis, we just like to use a success-failure algorithm [16]. It can be described as below. Given starting point x and step size h, if

$$F(x+h) < F(x)$$

the step will be called a success; otherwise it will be called a failure. In the case of a success, the step size h will be increased and replaced by $\vartheta * h$, and

$$x := x + h$$

where ϑ is called the success factor, and we try again. In the case of a failure, the step size will be reduced and h is replaced by $\tau * h$, and then we try again. Generally ϑ and τ can be set to 2.5 and 0.5 respectively, but they are application dependent. In this thesis these two values are set to 1.95 and 0.2 respectively.

This algorithm is very simple and easily implemented for neural networks since it only involves function evaluations. The function evaluations are equivalent to presenting input patterns to the input layer and passing them forward to the output layer, and then comparing this computed output with the desired output, resulting in an error that is the function value we desire. The calculation of $\frac{\partial E}{\partial w_{ij}}$ is equivalent to computing a search direction. And finally, the computation of the step size is equivalent to deciding a learning rate. It is necessary to point out that the error function is based on the entire training set, and the connection weights are updated after an entire set of training examples have been presented to the network. We call this training mode batch mode.

Chapter IV

Results and Analysis

In this chapter, we will give the results of comparison of the performance of two different neural networks as well as five different training algorithms for prediction of soil moisture content. These two networks are a feedforward fully connected neural network and a cascade correlation network, and the five algorithms are standard back-propagation, quick back-propagation, delta bar delta, extended delta bar delta, and steepest descent in batch mode with line search.

Test Data Preparation

To do the comparison of performance mentioned above, we use temperature data sampled from a field for one year. The depth at which the soil moisture content is to be predicted is chosen to 30 cm from the soil surface. As discussed in Chapter III, to predict soil moisture content at one point 30 cm deep, we need to know the temperatures at depths of 10, 20, 40, 50 cm respectively. For each day we use temperature and moisture content data at times 2 am and 2 pm. Since each level has three temperature sample sites, for each input pattern we have 24 temperature inputs and one bias that is permanently set to 1.0. To study how the network's performance behaves after training, we divide the whole data set into two parts: one is the training data set, the other is a test data set that is never exposed to the network during the training course, each of which

has 154 data points. The division into two data sets can be done by extracting temperature data of every other day into another set.

Before a training pattern is presented to the network, it needs to be normalized. There are some problems that can arise due to not normalizing the data before training. To normalize, we generate a MinMax table that contains the maximum and minimum value of each field of the entire training set. The normalization can carried out as below:

$$\text{scale} = \frac{(\text{high} - \text{low})}{(\max_i - \min_i)}$$

$$\text{offset} = \frac{\text{high} * \min_i - \text{low} * \max_i}{\max_i - \min_i}$$

$$\text{output}_{\text{scaled}} = \text{input} * \text{scale} + \text{offset}$$

where $\max_i$ and $\min_i$ are the maximum and minimum of field i through the whole training set; high and low are the range we would like to scale the input.

The initialization of neural networks also has an effect on the learning time [17]. Several methods have been invented to give neural networks as good an initial state as possible. This can be done by either some understanding of the learning mechanism in the networks or some prior knowledge [17]. We can initialize the network with random values uniformly distributed on [0,1].

Convergence Criterion

First we need to define the learning time. There are several definitions of learning time. One is number of the epochs, where an epoch is defined as one pass through the entire set of training examples [7]. But some researchers have defined an epoch as a subset of the entire training set [5]. In this study, we adopt the first definition. The other definition of learning time is simply the number of presentations of input patterns. In this thesis, we give both of them as a measure of learning time.

To set a convergence criterion, one popular method is to use RMS error [4], which is defined as below

$$RMS = \sqrt{\frac{\sum_i (d_i - o_i)^2}{N}}$$

where d_i is the actual output and o_i is the desired output. N is the number of presentations of input patterns. A desired maximum value of RMS is set to certain value before training begins. When the criterion RMS is satisfied, the training will stop. There are some misunderstandings that the poor generalization of a neural network from the training set to the test set results from overtraining. In many applications, many users have commonly overparameterized the network having the number of weights only a little less than the number of training examples or even larger than the number of training examples. This lead to overfitting of the training data and consequent poor generalization. Some users have tried to cure this by stopping training before reaching even a local minimum. This is not a reasonable solution. The correct solution is to reduce the number of weights of the network, or perhaps to use a smoothing or regularization approach [26]. There is a rule of thumb for obtaining good generalization

of a network trained by examples is that one should use the smallest network that will fit the training data [26]. Usually we want the number of weights of the network significantly less than the number of training examples.

Results

First we investigate the Cascade Correlation network. We start with a minimal structure for this network, that is, the original network consists only of the input layer and the output layer. At this time, it has 25 weights and no hidden units. As indicated before, the Cascade Correlation network will add new hidden nodes during the training course, one at a time. There are two parameters that govern the process of adding a new hidden node, one is outEpochs and the other is Threshold. The parameter outEpochs gives the maximum number of epochs to train the output layer before Threshold can be satisfied. After the maximum number of epochs has elapsed, a new hidden node can be added to the existing network. The parameter Threshold gives a criterion that will stop training the output layer if it is satisfied, and add a new hidden node. The convergence behavior of the Cascade Correlation network for prediction of soil moisture is given in Table I. The final architecture of this cascade correlation network consists of one hidden unit with 26 weights. It needed approximately 60 epochs of training to get to the RMS value of 0.03748.

Next we investigate three networks with standard back-propagation, which have one hidden layer with three, four or five units, and the numbers of weights of 79, 105, 131 respectively. The total number of nodes of each of these networks are 29, 30 and 31, including 24 input units, 3, 4 or 5

hidden units, one output unit, and one bias that is permanently connected to a constant input of 1.0. The convergence behavior of these networks are shown in Table II, III, IV. The networks with 3, 4 or 5 hidden units have no significant difference in terms of convergence speed and generalization. For the network with four hidden units, it needs approximately 150 epochs to get to an RMS value of 0.03831. Actually, we kept on training until the number of epochs reached 600, but there was no significant improvement.

For QuickProp, we use networks of the same architectures as in the standard back-propagation above. This means that we have total number of nodes of 29, 30, 31 each, the weights of 79, 105, 131 respectively, and one hidden layer with three, four or five units. We find that the networks with 3, 4 and 5 hidden units have almost the same convergence speed and generalization performance. This may suggest that when the number of hidden units of the network with QuickProp falls into some range, their convergence behavior and generalization performance will not be sensitive to the changes in the number of hidden units. In Table V, VI, VII for QuickProp, we can find that it is almost 5 times faster than standard back-propagation for solving the problem of prediction of soil moisture content. In Fahlman's experiment with the complement encoder problems, the QuickProp is about 6 times faster than the standard back-propagation. This shows that the QuickProp is a promising method for speeding up convergence of networks in wider applications.

The result of steepest descent in batch mode with line search is shown in Table VIII, IX, X. The networks with 3, 4 or 5 hidden units have almost the same convergence speed and generalization performance. We also use the same architecture as in the standard back-propagation above.

Since it updates the connection weights after all training patterns have been presented, extra storage is needed to hold the accumulated delta weights.

Table XI, XII, XIII and Table XIV, XV, XVI show the results of DBD and EDBD. Both of them use the same architecture as in standard back-propagation above. Both of the networks with 4 and 5 hidden units converge faster than the one with 3 hidden units for the DBD rule, but the network with 5 hidden units has poorer generalization performance than the one with 3 or 4 hidden units. This is due to the overparameterization of the network with the DBD rule. For the EDBD rule, the network with 3 hidden units has almost the same convergence speed as the ones with either 4 or 5 hidden units, but it has better generalization performance than both of them. From these tables above, we can see that DBD and EDBD are faster than standard back-propagation. This is due to changing their learning parameters dynamically. Since DBD needs to adjust dynamically each learning rate associated with each weight, it needs the same amount of storage to hold the time-varying rates as that of weights. So it requires twice as much storage as the standard back-propagation does. For EDBD, in addition to dynamically adjusting learning rates, it also needs to dynamically adjust the momentum term. So it requires three times as much storage as the standard back-propagation does.

The comparison of these training methods and the cascade network are summarized in Table XVII. We give some discussion about this table. The cascade method may be the best one of all method. It has the same order of convergence as the QuickProp and the steepest descent, but it only one hidden node. More important, since it adds hidden nodes dynamically during the training course, we don't have to worry about such

things as choosing the number of hidden layer as well as the number of units in each layer beforehand as in the case of feedforward fully-connected network. Therefore, some overparameterization can be avoided. QuickProp is faster than standard back-propagation because it considers not only the first derivative of error function E with respect to the weight w_i, but also the second derivative of E with respect to w_i. The speeding up of convergence of the network by DBD and EDBD was at the cost of adjusting the learning rates and momentum terms dynamically.

Table I

Convergence Behavior for a Cascade Correlation Network

# of epochs	# of iterations	# of hidden units	RMS
2	308	0	0.06488
4	616	0	0.2133
6	924	0	0.04920
8	1232	0	0.04789
10	1540	0	0.04180
12	1848	0	0.04162
14	2156	0	0.04009
16	2464	0	0.04031
18	2772	0	0.03941
20	3080	0	0.03927
22	3388	0	0.03907
24	3696	0	0.03865
26	4004	0	0.03852
28	4312	0	0.03851
30	3620	0	0.03833
40	6160	0	0.03804
50	7700	1	0.03789
55	8470	1	0.03809
60	9240	1	0.03748

Table II

Convergence behavior for Standard Back-Propagation with Three Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	3	00.2983
10	1540	3	0.03839
20	3080	3	0.03842
30	3620	3	0.03840
40	6160	3	0.04436
50	7700	3	0.04433
60	9240	3	0.04753
70	6160	3	0.03921
80	7700	3	0.04253
90	9240	3	0.04223
70	10780	3	0.03808
100	15400	3	0.03854
110	16940	3	0.03852
120	18480	3	0.03884
130	20020	3	0.04066
135	20790	3	0.03979
140	21560	3	0.03886
145	22330	3	0.03816
150	23100	3	0.03777

Table III

Convergence behavior for Standard Back-Propagation with Four Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
2	308	4	0.1024
4	616	4	0.06440
6	924	4	0.08151
8	1232	4	0.04146
10	1540	4	0.03941
20	3080	4	0.04173
30	4620	4	0.05043
40	6160	4	0.03844
50	7700	4	0.03981
60	9240	4	0.06605
70	10780	4	0.03808
80	12130	4	0.03808
90	13860	4	0.03797
100	15400	4	0.03795
110	16940	4	0.03796
120	18480	4	0.03800
130	19500	4	0.03810
140	21560	4	0.03822
150	23100	4	0.03831

Table IV

Convergence behavior for Standard Back-Propagation with Five Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	5	0.2786
10	1540	5	0.03905
20	3080	5	0.03900
30	4620	5	0.04025
40	6160	5	0.03879
50	7700	5	0.03873
60	9240	5	0.04225
70	10780	5	0.04057
80	12130	5	0.03800
90	13800	5	0.03822
95	14630	5	0.03847
100	15400	5	0.03927
105	16170	5	0.04039
110	16940	5	0.04030
115	17710	5	0.04840
120	18480	5	0.03943
125	19250	5	0.03795
130	19500	5	0.03790
135	20790	5	0.03793

Table V

Convergence Behavior for QuickProp Back-Propagation with Three Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	3	0.1849
2	308	3	0.1470
3	462	3	0.2061
4	616	3	0.06129
5	770	3	0.04438
6	924	3	0.03855
7	1078	3	0.03850
8	1232	3	0.03840
9	1386	3	0.03842
10	1540	3	0.03841
12	1848	3	0.06980
14	2156	3	0.04382
16	2464	3	0.03919
18	2772	3	0.03951
20	3080	3	0.03954
22	3388	3	0.03777
24	3696	3	0.03728
26	4004	3	0.03718
28	4312	3	0.03696

Table VI

Convergence Behavior for QuickProp Back-Propagation with Four Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	4	0.04514
2	308	4	0.03948
3	462	4	0.04010
4	616	4	0.03836
5	770	4	0.03836
6	924	4	0.03837
7	1078	4	0.03838
8	1232	4	0.03839
9	1386	4	0.08753
10	1540	4	0.05530
12	1848	4	0.07939
14	2156	4	0.03888
16	2464	4	0.03878
18	2772	4	0.03866
20	3080	4	0.03841
25	3850	4	0.03788
30	3620	4	0.03828
33	5082	4	0.03853
34	5236	4	0.03829

Table VII

Convergence Behavior for QuickProp Back-Propagation with Five Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	5	0.1783
2	308	5	0.07344
3	462	5	0.2138
4	616	5	0.09049
5	770	5	0.1024
6	924	5	0.05633
7	1078	5	0.1304
8	1232	5	0.04696
9	1386	5	0.06407
10	1540	5	0.03885
12	1848	5	0.03851
14	2156	5	0.03863
16	2464	5	0.03899
18	2772	5	0.05017
20	3080	5	0.04056
22	3388	5	0.03844
24	3896	5	0.03833
26	4004	5	0.03801
29	4466	5	0.03723

Table VIII

Convergence Behavior for Steepest Descent in Batch Mode with Line Search with Three Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	3	0.07044
2	308	3	0.06825
3	462	3	0.06387
4	616	3	0.05992
5	770	3	0.05636
6	924	3	0.05323
7	1078	3	0.05057
8	1232	3	0.04834
9	1386	3	0.04650
10	1540	3	0.04499
12	1848	3	0.04275
14	2156	3	0.04124
16	2464	3	0.04021
18	2772	3	0.03949
20	3080	3	0.03895
22	3388	3	0.03859
24	3696	3	0.03831
26	4004	3	0.03809
30	3620	3	0.03795

Table IX

Convergence Behavior for Steepest Descent in Batch Mode with Line Search with Four Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	4	0.07425
2	308	4	0.07253
3	462	4	0.06790
4	616	4	0.06373
5	770	4	0.05998
6	924	4	0.05659
7	1078	4	0.05358
8	1232	4	0.05095
9	1386	4	0.04873
10	1540	4	0.04687
12	1848	4	0.04404
14	2156	4	0.04211
16	2464	4	0.04078
18	2772	4	0.03985
20	3080	4	0.03919
22	3388	4	0.03871
24	3696	4	0.03836
26	4004	4	0.03810
28	4312	4	0.03791

Table X

Convergence Behavior for Steepest Descent in Batch Mode with Line Search with Five Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	5	0.07702
2	308	5	0.07561
3	462	5	0.07052
4	616	5	0.06258
5	770	5	0.06173
6	924	5	0.05798
7	1078	5	0.05466
8	1232	5	0.05179
9	1386	5	0.04936
10	1540	5	0.04733
12	1848	5	0.04426
14	2156	5	0.04218
16	2464	5	0.04075
18	2772	5	0.03977
20	3080	5	0.03808
22	3388	5	0.03857
24	3696	5	0.03823
26	4004	5	0.03797
28	4312	5	0.03787

Table XI

Convergence Behavior for DBD with Three Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	3	0.05201
2	308	3	0.03870
3	462	3	0.03892
4	616	3	0.03819
5	770	3	0.04039
6	924	3	0.04047

Table XII

Convergence Behavior for DBD with Four Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	4	0.05219
2	308	4	0.03685
3	462	4	0.03610
4	616	4	0.03645
5	770	4	0.03091
6	924	4	0.03085
7	1078	4	0.02963
8	1232	4	0.03007
9	1386	4	0.02826
10	1540	4	0.02872
11	1694	4	0.02949
12	1848	4	0.02964
13	2002	4	0.02845
14	2156	4	0.03142

Table XIII

Convergence Behavior for DBD with Five Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	5	0.05050
2	308	5	0.03702
3	462	5	0.03629
4	616	5	0.03236
5	770	5	0.03137
6	924	5	0.02971
7	1078	5	0.03148
8	1232	5	0.02908
9	1386	5	0.03889
10	1540	5	0.03013

Table XIV

Convergence Behavior for EDBD with Three Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	3	0.06901
2	308	3	0.04667
3	462	3	0.04140
4	616	3	0.03952
5	770	3	0.04469
6	924	3	0.04133
7	1078	3	0.03901
8	1232	3	0.03808
9	1386	3	0.03764
10	1540	3	0.03643
11	1964	3	0.04202
12	1848	3	0.03827
13	2002	3	0.03780
14	2156	3	0.03698
15	2310	3	0.03669
16	2464	3	0.03733
17	2618	3	0.03695
18	2772	3	0.03618
19	2926	3	0.03611

Table XV

Convergence Behavior for EDBD with Four Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	4	0.06789
2	308	4	0.04491
3	462	4	0.04086
4	616	4	0.03876
5	770	4	0.03775
6	924	4	0.03699
7	1078	4	0.04209
8	1232	4	0.03931
9	1386	4	0.04036
10	1540	4	0.03876
11	1964	4	0.03659
12	1848	4	0.03682
13	2002	4	0.03793
14	2156	4	0.03695
15	2310	4	0.03678
16	2464	4	0.03658
17	2618	4	0.04087
18	2772	4	0.03857
19	2926	4	0.03465

Table XVI

Convergence Behavior for EDBD with Five Hidden Units

# of epochs	# of iterations	# of hidden units	RMS
1	154	5	0.065859
2	308	5	0.04370
3	462	5	0.03917
4	616	5	0.03699
5	770	5	0.03612
6	924	5	0.03792
7	1078	5	0.03663
8	1232	5	0.03656
9	1386	5	0.03807
10	1540	5	0.03809
11	1964	5	0.03660
12	1848	5	0.03543
13	2002	5	0.03569
14	2156	5	0.03519
15	2310	5	0.03470
16	2464	5	0.03658
17	2618	5	0.03473
18	2772	5	0.03532
19	2926	5	0.03421

Table XVII

Comparison of Convergence Behavior for Different Architecture and Algorithms

alg. or arch.	# hidden units	# weights	# of epochs	# of iterations	training RMS	testing RMS
cascade	1	26	60	9240	0.03748	0.03810
std BP	3	79	155	23870	0.03772	0.03815
std BP	4	101	150	23100	0.03831	0.03869
std BP	5	131	135	20790	0.03793	0.03835
quick BP	3	79	28	4312	0.03696	0.03750
quick BP	4	101	34	5236	0.03829	0.03870
quick BP	5	131	29	4466	0.03723	0.03777
SD	3	79	30	3620	0.03795	0.03810
SD	4	101	34	4466	0.03791	0.03783
SD	5	131	27	4158	0.03787	0.03805
DBD	3	79	7	1078	0.04047	0.04366
DBD	4	101	15	2310	0.03142	0.03940
DBD	5	131	11	1694	0.03013	0.04366
EDBD	3	79	19	2926	0.03611	0.03770
EDBD	4	101	20	3080	0.03465	0.03912
EDBD	5	101	19	2926	0.03421	0.037530

Chapter V

Conclusions

We studied two kinds of neural networks: a feedforward fully-connected network and a cascade correlation network for prediction of soil moisture content. The comparison of performance of five training methods with a fully-connected network and a cascade network was made. By experimental results, we can get following conclusions:

- Standard back-propagation is the slowest of all methods.
- QuickProp is faster than standard back-propagation.
- Cascade correlation has the same order of convergence as the QuickProp, but it needs fewer hidden units than a fully-connected feedforward network, resulting in less storage requirement for connection weights, and is less prone to overparameterization.
- Steepest descent in batch mode with line search is as fast as QuickProp, but it needs extra storage to hold accumulated delta weights than general incremental methods.
- DBD and EDBD have almost the same convergence speed, and both of them are faster than QuickProp and steepest descent in batch mode with line search.

A Selected Bibliography

[1] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation", Neural Networks, Vol. 1, pp. 295-307, 1988.

[2] A. A. Minai and R. D. Williams, "Acceleration of Back-Propagation through learning rate and momentum adaptation", International Joint Conference on Neural networks, Vol. I, pp. 676-679, Jan. 1990.

[3] C. T. Altendorf, M. L. Stone and R. L. Elliott, "Using a neural network for soil moisture prediction", 1992 International Winter Meeting of the American Society of Agricultural Engineers.

[4] NeuralWare Manual, NeuralWare, Inc., Pittsburgh, PA, 1991

[5] Scott E. Fahlman, "An empirical study of learning speed in Back-Propagation networks", CMU Technical Report, CMU-CS-88-162, June 1988.

[6] P. W. Jones and J. Hoskins, "Back-propagation--A generalized delta learning rule", BYTE Magazine, Oct. 1987.

[7] S. E. Fahlman and C. Lebiere, "The Cascade-Correlation learning architecture", School of Computer Science report CMU-CS-90-100, Pittsburgh, PA, Carnegie Mellon University.

[8] E. Barnard, "Optimization for training neural nets", IEEE Trans. Neural Networks, Vol. 3, pp. 232-240, Mar 1992.

[9] E. Davalo, P Naim, Neural Network, Macmillan Education Ltd., 1991.

[10] V. D. Dmagt, "Minimization method for training feedforward neural networks", Neural Networks, Vol. 7, No 1, pp. 1-11, 1994

[11] H. Drucher and Y. l. Cun, " Improving generalization performance using double propagation", IEEE Transactions on Neural Networks, Vol. 3, Now. 6, November 1993

[12] S. E. Fahlmann and E. Hiton, " Connectionist architectures for artificial intelligence", Computer, pp. 100-108, January 1987

[13] J. D. Villiers and E. Barnard, " Back propagation neural net with one and two hidden layers", IEEE Transactions on Neural Networks, Vol. 4, No. 1, Januaray 1992

[14] A. C. Eaton and T. L. Oliver, "Learning Coefficient dependence on training set size", Neural Networks, Vol. 5, pp. 283-288, 1992

[15] M. F. Moller, " A scaled conjugate gradient algorithm for fast supervised learning", Neural Networks, Vol. 6, pp. 525-533, 1993

[16] A. Sperduti and A. Starita, "Speed up learning and network optimization with extended back propagation", Neural Networks, Vol. 6, pp. 365-383, 1993

[17] T. Denoeux and R. Lengelle, " Initialize back-propagation networks with prototypes", Neural networks, Vol. 6, pp. 351-363, 1993

[18] M. Hoehfield and S. E. Fahlman, " Learning with limited numerical precision using the cascade correlation algorithm", IEEE Transactions on Neural Networks, Vol. 3, No. 4, July 1992

[19] M. A. Andree, G. T. Barkema, W. Lourrens, and A. Teal, " A comparison study of binary feedforward neural networks and digital circuits", Neural networks, Vol. 6, pp. 785-790, 1993

[20] A. V. Ooyen, "Improving the convergence of the back-propagation algorithm", Neural Networks, Vol. 5, pp. 465-471, 1992

[21] M. S. Cheng, " A survey and comparison of conjugate gradient methods of optimization", Oklahoma State University Master Thesis, 1993

[22] D. E. Rumelhart, G. E. Hinton and R. J. Williams, " Learning representations by back propagating errors", Nature, No. 323, pp. 533-536, 1986

[23] T. P. Vogl, "Accelerating the convergence of the back-propagation method", Biological Cybernetics, Vol. 59, pp. 259-263, 1988

[24] J. Kowalik & M. R. Osborne, Methods for Unconstrained Optimization, Elsevier, 1968

[25] J. E. Dennis & Jr. Robert, Numerical Methods for Unconstrained Optimization and Nonlinear Equations, Prentice-Hall, 1983

[26] R. Reed, "Pruning algorithm -- a survey", IEEE Transaction on Neural Networks, Vol. 4, No. 5, 1993

APPENDIX A

```
C THIS PROGRAM IMPLEMENTS THE NEURAL NETWORKS WITH
C ALGORITHMS OF STANDARD BACK PROPAGATION, STEEPEST
C DSCENT IN BATCH MODE WITH LINE SEARCH, DELTA
C BAR DELTA, EXTENDED DELTA BAR DELTA.
C THE TRAINING DATA IS KEPT IN A FILE CALLED "TRAIN.DAT"
C AND TEST DATA KEPT IN A FILE OF "TEST.DAT", WHICH IS
C NEVER EXPOSED TO THE NEURAL NETWORKS DURING THE TRAINING
C COURSE. AFTER THE TRAINING OF NEURAL NETWORKS HAS BEEN
C COMPLETED, IT WILL BE TESTED USING TEST DATA AND THE RESULT
C WILL BE STORED IN A FILE OF TEST.NNR.
C
      INTEGER I,J,II
C DEFINITION OF CONNECTION WEIGHTS, INPUT NODES, OUTPUT
C NODES, TIME VARYING PARAMETER MATRIX
      REAL MOISTURE,EPSILON,SUMERR
     *,NODEIN(30,30),NODEOUT(30,30),W(3,30,30),DW(3,30,30)
     *,DELTA(3,30,30),ALPHA(3,30,30),DWA(3,30,30),TEMP(310,26)
     *,MIN(25),MAX(25),RI,WM,MU(3,30,30)
C SCALE PARAMETER, TRAINING PARAMETER
      REAL HIGH,LOW,OFFSET,SCALE,YI,SI,YIO,SIO
      INTEGER NMNODE(3),TRMD,EPOCH,EPLENGTH,
     *NN,EPCNT,JJ
     *,INL,HHDL,OUTL,FRT,LST,NDPT,FLAG,MAXEP,OUT1,OUT2
C
C INPUT LAYER NUMBER
      INL=1
C HIDDEN LAYER NUMBER
      HHDL=2
C OUTPUT LAYER NUMBER
      OUTL=3
C NUMBER OF NODES IN INPUT LAYER
      NMNODE(INL)=25
C NUMBER OF HIDDEN NODES
      NMNODE(HHDL)=4
```

```
C NUMBER OF OUTPUT NODE
    NMNODE(OUTL)=1
C NUMBER OF LAYERS
    LAYER=3
C NUMBER OF HIDDEN UNITS
    N=NMNODE(HHDL)
C NUMBER OF INPUT UNITS
    M=NMNODE(INL)
C OUTPUT RANGE
    HIGH=1.0
    LOW=0.0
C CONVERGENCE CRITERION
    EPSILON=0.037
C SUM SQUARED ERROR INITIALIZATION
    SUMERR=0.0
C THE COUNTER FOR EPOCHS ELAPSED
    EPCNT=0
C NUMBER OF DATA POINTS IN THE TRAINING SET
    NDPT=154
C READ FLAG: FLAG=1 READ TRAINING FILE
C FLAG=2 READ FROM TESTING FILE
    FLAG=1
C MAXIMUM NUMBER OF EPOCHS SET
    MAXEP=1000
C THE NUMBER OF PRESENTATIONS TO UPDATA WEIGHTS
C IF EPLENGTH=1--A INCREMENTAL UPDATE WEIGHTS
    EPLENGTH=1
C THE COUNTER FOR UPDATE WEIGHT
    EPOCH = 0
C MAXIMUM OF WEIGHT
    WM=100.0
C OUTPUT DEVEICE NUMBER
    OUT1=4
    OUT2=6
    OUT3=5
C TRAINING METHOD: TRMD=1--STANDARD BACK-PROPAGATION
```

```
C TRMD=3--DELTA BAR DELTA; TRMD=4--EXTENDED DELTA BAR DELTA
    TRMD=4
C THE COUNTER FOR COMPUTE RMS ERROR
    NN = 0
C
C OPEN PARAMETER FILE
C     OPEN(OUT3,FILE='par.dat')
C THE TRAINING RMS FILE
    OPEN(OUT2,FILE='err.dat')
C INPUT TEMPERATURE DATA
    CALL RDINPUT(TEMP,NDPT,NMNODE(INL),FLAG,LST)
C RANDOMIZE CONNECTION WEIGHTS
    CALL  RANWT(W,NMNODE,LAYER,INL,HHDL)
C COMPUTE MIN-MAX TABLE
    CALL MNTAB(TEMP,MIN,MAX,NDPT,LST)
C INITIALIZE DELTA WEIGHTS
    CALL INITDW(DW,DELTA,ALPHA,NMNODE,INL,HHDL)
 50  RI=RAND()
C IF NUMBER OF EPOCHS OF TRAINING LARGER THAN MAXIMUM
C THEN STOP TRAINING
    IF (EPCNT .EQ. MAXEP) THEN
      GOTO 32
    ENDIF
    JJ=JJ+1
    II=MOD(JJ,NDPT)
    IF (II .EQ. 0) THEN
      II=1
    ENDIF

    DO 20 J=1,NMNODE(INL)
      NODEOUT(INL,J)=TEMP(II,J)
      IF(NODEOUT(INL,J) .EQ. 0.0 )THEN
        STOP
      ENDIF
 20  CONTINUE
C SET INPUT RANGE BETWEEN 0 AND 1.0
```

```
C NORMALIZATION OF INPUT TEMPERATURE DATA
   SIO=NODEOUT(FRT,NMNODE(INL))
   DO I=1,NMNODE(INL)
     SCALE=(HIGH-LOW)/(MAX(I)-MIN(I))
     OFFSET=(MAX(I)*LOW-MIN(I)*HIGH)/(MAX(I)-MIN(I))
     NODEOUT(FRT,I)=NODEOUT(FRT,I)*SCALE+OFFSET
   END DO
C FORWARD INPUT TRAINING PATTERN TO OUTPUT LAYER
   CALL FDINPUT(W,NODEIN,NODEOUT,NMNODE,LAYER,N,M)
C ACTUAL OUTPUT FROM SAMPLE DATA SCALED TO 0 AND +1.0
C    YIO=NODEOUT(LAYER,FRT)
C    NODEOUT(LAYER,FRT)= NODEOUT(LAYER,FRT)*SCALE+OFFSET
   SI=NODEOUT(FRT,LST)
C PREDICTED OUTPUT SCALED TO ACTUAL VALUE OF 0 AND 0.4
   YI=NODEOUT(LAYER,FRT)
   SCALE=(MAX(LST)-MIN(LST)/(HIGH-LOW)
   OFFSET=(HIGH*MIN(LST)-LOW*MAX(LST)/(HIGH-LOW)
   YIO=YI*SCALE+OFFSET
C    WRITE(*,23) YIO,SIO
 23  FORMAT(1X,2F8.6)
C STANDARD BACK PROPAGATION ALGORITHM
   IF(TRMD .EQ. 1) THEN
     CALL STDBP(W,DW,NODEIN,NODEOUT,NMNODE,LAYER,N,M,
   *SI,YI)
   ENDIF
C DELTA BAR DELTA ALGORITHM
   IF(TRMD .EQ. 3)THEN
     CALL DBDBP(W,DW,NODEIN,NODEOUT,NMNODE,LAYER,N,M,
   *SI,YI,ALPHA,DELTA)
   ENDIF
C EXTENDED DELTA BAR DELTA ALGORITHM
   IF(TRMD .EQ. 4)THEN
     CALL EDBDBP(W,DW,NODEIN,NODEOUT,NMNODE,LAYER,N,M,
   *SI,YI,ALPHA,DELTA,MU)
   ENDIF
```

```
C     WRITE(OUT3,29) (ALPHA(LAYER,FRT,I),I=1,NMNODE(HHDL))
C     WRITE(OUT3,29) (MU(LAYER,FRT,I),I=1,NMNODE(HHDL))
 29   FORMAT(1X,6F8.6)
      DO I=1,NMNODE(HHDL)+1
        DWA(LAYER,FRT,I)=DWA(LAYER,FRT,I)+DW(LAYER,FRT,I)
      END DO
      DO 25 I=1,NMNODE(2)
        DO 28 J=1,NMNODE(1)
          DWA(HHDL,I,J)=DWA(HHDL,I,J)+DW(HHDL,I,J)
 28     CONTINUE
 25   CONTINUE
      EPOCH=EPOCH+1
      IF(EPOCH .EQ. EPLENGTH)THEN
        EPOCH=0
C IF CONNECTION WEIGHT LARGER THAN BOUND
C SET IT TO THE BOUND
        DO I=1,NMNODE(HHDL)
C UPDATA WEIGHT
          W(LAYER,FRT,I)=W(LAYER,FRT,I)+DWA(LAYER,FRT,I)
          IF(W(LAYER,FRT,I) .GT. WM)THEN
            W(LAYER,FRT,I)=WM
          ENDIF
          IF(W(LAYER,FRT,I).LT. -WM)THEN
            W(LAYER,FRT,I)=-WM
          ENDIF
        ENDDO
        DO 40 I=1,NMNODE(HHDL)
          DO 42 J=1,NMNODE(INL)
C UPDATE THE WEIGHT
            W(HHDL,I,J)=W(HHDL,I,J)+DWA(HHDL,I,J)
            IF(W(HHDL,I,J) .GT. WM )THEN
              W(HHDL,I,J)=WM
            ENDIF
            IF(W(HHDL,I,J) .LT. -WM)THEN
              W(HHDL,I,J)= -WM
            ENDIF
```

```
42        CONTINUE
40      CONTINUE
C RESET DELTA WEIGHT
        DO I=1,NMNODE(HHDL)
        DWA(LAYER,FRT,I)=0.0
        END DO
        DO 26 I=1,NMNODE(HHDL)
          DO 27 J=1, NMNODE(INL)
            DWA(HHDL,I,J)=0.0
27        CONTINUE
26      CONTINUE
      ENDIF
C COMPUTE SUM SQUARED ERROR
      SUMERR=SUMERR+(SIO-YIO)*(SIO-YIO)
      IF(NN .EQ. NDPT)THEN
        NN=0
C SUM SQUAREED ROOT ERROR
        SUMERR=SQRT(SUMERR/NDPT)
        EPCNT=EPCNT+1
C OUTPUT SUM SQUARED ERROR AND NUMBER OF EPOCHS ELAPSED
        WRITE(OUT2,31) EPCNT,SUMERR
        WRITE(*,31) EPCNT,SUMERR
31      FORMAT(1X,I5,' ',F8.6)
        IF (SUMERR .LT. EPSILON) THEN
C IF TRAINING COMPLETED, START TO TEST NETWORKS
C READ TEST DATA
        FLAG=2
32      CALL RDINPUT(TEMP,NDPT,NMNODE(INL),FLAG)
C OPEN TEST RESULT FILE
        OPEN(OUT1,FILE='test.nnr')
        WRITE(OUT1,45) COUNTER
        DO K=1,NDPT
          DO 21 J=1,NMNODE(INL)
            NODEOUT(FRT,J)=TEMP(K,J)
            IF(NODEOUT(INL,J) .EQ. 0.0 )THEN
              STOP
```

```
          ENDIF
 21       CONTINUE
C
        SIO=NODEOUT(FRT,LST)
C NORMALIZATION OF INPUT TEPERATURE DATA
        DO 22 I=1,NMNODE(INL)
          SCALE=(HIGH-LOW)/(MAX(I)-MIN(I))
          OFFSET=(MAX(I)*LOW-MIN(I)*HIGH)
     *    /(MAX(I)-MIN(I))
          NODEOUT(FRT,I)=NODEOUT(FRT,I)*SCALE+OFFSET
 22       CONTINUE
C FORWARD INPUT DATA TO OUTPUT LAYER
        CALL FORWARDFDINPUTINPUT(W,NODEIN,NODEOUT,NMNODE,LAYER,N,M)
        YI=NODEOUT(LAYER,FRT)
C SCALED BACK TO ACTUAL RANGE OF TEMPERATURE
        SCALE=(MAX(LST)-MIN(LST))/(HIGH-LOW)
        OFFSET=(HIGH*MIN(LST)-LOW*MAX(LST))/(HIGH-LOW)
        YIO=YI*SCALE+OFFSET
        WRITE(OUT1,23) YIO,SIO
      ENDDO
      CLOSE(OUT1)
      CLOSE(OUT3)
      CLOSE(OUT2)
      STOP
      ENDIF
C IF CONVERGENCE CRITERION NOT SATISFIED, CONTINUE TRAINING
      IF(SUMERR .GE. EPSILON)THEN
        NN = 0
        SUMERR=0.0
        GOTO 50
      ENDIF
    ENDIF
    IF(NN .LT. NDPT)THEN
      NN=NN+1
      GOTO 50
    ENDIF
```

```
 45  FORMAT(I5)
 60  END
C
C
C***************************************************************
     SUBROUTINE FDINPUT(W,NODEIN,NODEOUT,NMNODE,LAYER,N,M)
C FORWARD INPUT VECTOR TO OUTPUT LAYER
     REAL W(LAYER,N,M),NODEIN(N,M),NODEOUT(N,M)
     INTEGER NMNODE(M),N,M
     INTEGER I,J,K
     REAL SUM,SCALE,OFFSET
C SUM -- THE SUM OF ALL INPUT TO ONE NODE
C
     DO 100 I=2,LAYER
       DO 110 J=1,NMNODE(I)
         SUM=1.0*W(I,J,1)
         DO 120 K=2,NMNODE(I-1)
C SUM ALL INPUT WITH CONNECTION WEIGHTS
             SUM=SUM+NODEOUT(I-1,K-1)*W(I,J,K)
 120        CONTINUE
        NODEIN(I,J)=SUM
C ACTIVATION FUNCTION TO GET OUTPUT
        NODEOUT(I,J)=FUN(SUM)
 110     CONTINUE
 100  CONTINUE
     END
C
C
C***************************************************************
     SUBROUTINE STDBP(W,DW,NODEIN,NODEOUT,NMNODE,LAYER,N,M,SI,YI
    *,INL,HHDL,OUTL,LCOEF)
C STANDARD BACKPROPAGATION ALGOTITHM
C INPUT: DESIRED OUTPUT, ACTUAL OUTPUT
C CONNECTION WEIGHTS AND INPUT AND OUTPUT NODES
     REAL W(LAYER,N,M),DW(LAYER,N,M),NODEIN(N,M),NODEOUT(N,M),
    *SI,YI
```

```
      INTEGER NMNODE(N),N,M,LAYER
      REAL DI,DJ,LCOEF
      INTEGER I,J,INL,HHDL,OUTL
C
C OUTPUT LAYER COMPUTATION OF DELTA WEIGHTS
      DI=(SI-YI)*FUND(NODEIN(LAYER,FRT))
      DW(LAYER,FRT,FRT)=LCOEF*DI*FUN(W(LAYER,INL,FRT))
      DO 200 I=1, NMNODE(LAYER-1)
       DW(LAYER,FRT,I+1)=LCOEF*DI*NODEOUT(LAYER-1,I)
 200  CONTINUE
C HIDDEN LAYER DELTA WEIGHTS COMPUTATION
      DO 210 I=1,NMNODE(LAYER-1)
       DJ=DI*W(LAYER,FRT,I)*FUND(NODEIN(HHDL,I))
       DW(HHDL,I,FRT)=LCOEF*DJ*FUN(W(HHDL,I,FRT))
       DO 220 J=1,NMNODE(LAYER-2)
         DW(HHDL,I,J+1)=LCOEF*DJ*NODEOUT(LAYER-2,J)
 220    CONTINUE
 210  CONTINUE
      RETURN
      END
C
C
C*****************************************************************
      SUBROUTINE DBDBP(W,DW,NODEIN,NODEOUT,NMNODE,LAYER,N,M,SI,YI
     *DELTA,ALPHA,INL,HHDL,OUTL,LCOEF)
C DELTA BAR DELTA ALGORITHM
C INPUT: DESIRED OUTPUT AND ACTUAL OUTPUT AS WELL
C AS CONNECTION WEIGHTS, INPUT AND OUTPUT NODES
C OUTPUT: COMPUTE DELTA WEIGHT
C CONNECTION WEIGHT, DELTA WEIGHT, ALPHA PARAMETER
      REAL W(LAYER,N,M),DW(LAYER,N,M),DELTA(LAYER,N,M),ALPHA(LAYER,N,M)
     *,SI,YI,NODEIN(N,M),NODEOUT(N,M)
      INTEGER NMNODE(N),N,M
      REAL DI,DI1,DJ,DJ1,DELTAV,DELTAP,KI,THETA,PHI,ALPHAMAX
     *,LCOEF,DWMAX,PU
C
```

```
C   DEFINITION OF CONSTANTS OF LEARNING COEFICIENTS
C   AVERAGE FACTOR OF DLETA E
    THETA=0.1
C DELTA RULE PARAMETER
    KI=0.3
C   EXPONENTIAL FACTOR
    PHI=0.1
C   THE BOUND FOR ALPHA
    ALPHAMAX=0.8
C MAXIMUM DELTA WEIGHT
    DWMAX=15.0
C DELTA RULE PARAMETER
    PU=0.1
C
C OUTPUT LAYER DELTA WEIGHT COMPUTATION
    DI=(SI-YI)*FUND(NODEIN(LAYER,FRT))
    DI1=DI*FUN(1.0)
    DELTAV=(1-THETA)*DI1+THETA*DELTA(LAYER,FRT,FRT)
C IF DELTA AND DELTA AVERAGE HAVE SAME SIGNS
    IF(DELTAV * DELTA(LAYER,FRT,FRT) .GT. 0.0)THEN
      DELTAP=KI
    ENDIF
C IF DELTA AND DELTA AVERAGE HAVE DIFFERENT SIGNS
    IF(DELTAV*DELTA(LAYER,FRT,FRT) .LT. 0.0)THEN
      DELTAP=-PHI*ALPHA(LAYER,FRT,FRT)
    ENDIF
C IF DELTA EQUALS ZERO OR DELTA AVERAGE EQUALS ZERO
    IF(DELTAV*DELTA(LAYER,FRT,FRT) .EQ. 0.0) THEN
      DELTAP=0.0
    ENDIF
    DELTA(LAYER,FRT,FRT)=DI1
    ALPHA(LAYER,FRT,FRT)=ALPHA(LAYER,FRT)+DELTAP
C GIVE THE UPPER BOUND OF ALPHA PARAMETER
C IF COMPUTED ALPHA LARGER THAN THE UPPER BOUND
C SET ALPHA TO THAT BOUND
    IF (ALPHA(LAYER,FRT,FRT) .GT. ALPHAMAX)THEN
```

```
          ALPHA(LAYER,FRT,FRT)=ALPHAMAX
        ENDIF
        DW(LAYER,FRT,FRT)=ALPHA(LAYER,FRT,FRT)*DI1
C       IF (DW(LAYER,FRT,FRT) .GT. DWMAX)THEN
C         DW(LAYER,FRT,FRT)=PU
C       ENDIF
C
        DO 400 I=1,NMNODE(LAYER-1)
         DI1=DI*NODEOUT(LAYER-1,I)
C COMPUTE DELTA AVERAGE
         DELTAV=(1-THETA)*DI1+THETA*DELTA(LAYER,FRT,I+1)
C IF DELTA AND DELTA AVERAGE HAVE SAME SIGNS
         IF(DELTAV * DELTA(LAYER,FRT,I+1) .GT. 0.0)THEN
           DELTAP=KI
         ENDIF
C IF DELTA AND DELTA AVERAGE HAVE OPPSITE SIGNS
         IF(DELTAV*DELTA(LAYER,FRT,I+1) .LT. 0.0)THEN
            DELTAP=-PHI*ALPHA(LAYER,FRT,I+1)
         ENDIF
C IF DELTA EQUALS ZERO
         IF(DELTAV*DELTA(LAYER,FRT,I+1) .EQ. 0.0 )THEN
           DELTAP=0.0
         ENDIF
         DELTA(LAYER,FRT,I+1)=DI1
         ALPHA(LAYER,FRT,I+1)=ALPHA(LAYER,FRT,I+1)+DELTAP
C SET UPPER BOUND TO ALPHA
C IF ALPHA LARGER THAN  UPPER BOUND, SET ALPHA TO THE BOUND
         IF(ALPHA(LAYER,FRT,I+1) .GT. ALPHAMAX)THEN
           ALPHA(LAYER,FRT,I+1)=ALPHAMAX
         ENDIF
         DW(LAYER,FRT,I+1)=ALPHA(LAYER,FRT,I+1)*DI1
C        IF(ABS(DW(LAYER,1,I+1)) .GT. DWMAX)THEN
C          DW(LAYER,FRT,I+1)=PU
C        ENDIF
 400  CONTINUE
C THE FOLLOWING IS THE SAME AS ABOVE EXCEPT FOR HIDDEN LAYER
```

```
   DO 410 I=1,NMNODE(LAYER-1)
    DJ=DI*W(LAYER,FRT,I)*FUND(NODEIN(HHDL,I))
    DO 420 J=1,NMNODE(LAYER-2)
     DJ1=DJ * NODEOUT(LAYER-2,J)
C COMPUTE DELTA AND DELTA AVERAGE
     DELTAV=(1.0-THETA)*DJ1+THETA*DELTA(HHDL,I,J+1)
C IF DELTA AND DELTA AVERAGE HAVE SAME SIGNS
     IF(DELTA(HHDL,I,J+1) * DELTAV .GT. 0.0)THEN
      DELTAP=K1
     ENDIF
C IF DELTA AND DELATA AVERAGE HAVE OPPOSITE SIGNS
     IF(DELTA(HHDL,I,J+1)*DELTAV .LT. 0.0)THEN
      DELTAP=-PHI*ALPHA(HHDL,I,J+1)
     ENDIF
C IF DELTA OR DELTA AVERAGE EQUALS ZERO
     IF(DELTA(HHDL,I,J+1) * DELTAV .EQ. 0.0)THEN
      DELTAP=0.0
     ENDIF
     ALPHA(HHDL,I,J+1)=ALPHA(HHDL,I,J+1)+DELTAP
C SET UPPER BOUND TO ALPHA
C IF COMPUTED ALPHA IS LARGER THAN UPPER BOUND
C THEN SET ALPHA TO UPPER BOUND
     IF(ALPHA(HHDL,I,J+1) .GT. ALPHAMAX)THEN
      ALPHA(HHDL,I,J+1)=ALPHAMAX
     ENDIF
     DELTA(HHDL,I,J+1)=DJ1
     DW(HHDL,I,J+1)=ALPHA(HHDL,I,J+1)*DJ1
C       IF(ABS(DW(HHDS,I,J+1)) .GT. DWMAX)THEN
C        DW(HHDL,I,J+1)=PU
C       ENDIF
 420  CONTINUE
 410 CONTINUE
   RETURN
   END
C
C
```

```
C*****************************************************************
    SUBROUTINE EDBDBP(W,DW,NODEIN,NODEOUT,NMNODE,LAYER,N,M,SI,YI
   *,ALPHA,DELTA,MU,INL,HHDL,OUTL,LCOEF)
C EXTENDED DELTA BAR DELTA ALGORITHM
C EXTENDED DELTA BAR DELTA IS A MODIFICATION VERSION OF
C DELTA BAR DELTA WITH MOMENTUM BEING TIME-VARING
C CONNECTION WEIGHT, DELTA WEIGHT, INPUT NODE AND OUTPUT NODE
C ALPHA AND DELTA PARAMETER
    REAL W(LAYER,N,M),DW(LAYER,N,M),NODEIN(N,M),NODEOUT(N,M),SI,YI
    REAL DELTA(LAYER,N,M),ALPHA(LAYER,N,M),MU(LAYER,N,M)
    INTEGER N,M,NMNODE(N)
    REAL DI,DJ,DI1,DJ1
    INTEGER I,J
C DEFINITION OF DIFFERENT PARAMETERS FOR EDBD RULE
    REAL
DELTAP,DELTAV,DELTAMU,THETA,KALPHA,KMU,GALPHA,GMU,PALPHA,PMU
   *,ALPHAMAX,MUMAX,DWMAX,PU
C
    THETA = 0.1
C   CONSTANT LEARNING RATE SCALE FACTOR
    KALPHA = 0.2
C   CONSTATN MOMENTUM SCALE FACTOR
    KMU = 0.1
C   CONSTANT LEARNING RATE EXPONENTIAL FACTOR
    GALPHA = 0.05
C   CONSTANT MOMENTUM RATE EXPONENTILA FACTOR
    GMU = 0.01
C   CONSTANT LEARNING RATE DECREMENT FACTOR
    ALPHA = 0.1
C   CONSTANT MOMENTTUM RATE DECREMENT FACTOR
    PMU = 0.1
C   UPPER BOUND ON THE LEARNING RATE
    ALPHAMAX = 0.1
C   UPPER BOUND IN THE MOMENTUM RATE
    MUMAX = 0.01
C   MAXIMU VALUE OF DELTA WEIGHT
```

```
      DWMAX = 5
C     SET DELTA WEIGHT TO THIS VALUE IF LARGER THAN BOUND
      PU = 0.1
C
C COMPUTATION OF OUTPUT LAYER
      DI=(SI-YI)*FUND(NODEIN(LAYER,1))
      DI1=DI*FUN(W(LAYER,INL,FRT))
C COMPUTE DELTA AVERAGE
      DELTAV=(1.0-THETA)*DI1+THETA*DELTA(OUTL,FRT,FRT)
C IF DELTA AND DELTA AVERAGE HAVE SAME SIGNS
      IF(DELTAV*DELTA(OUTL,FRT,FRT) .GT. 0.0)THEN
        DELTAP=KALPHA*EXP(-GALPHA*ABS(DELTAV))
        DELTAMU=KMU*EXP(-GMU*ABS(DELTAV))
      ENDIF
C IF DELTA AND DELTA AVERAGE HAVE OPPOSITE SIGNS
      IF (DELTAV*DELTA(OUTL,FRT,FRT) .LT. 0.0)THEN
        DELTAP=-PALPHA*ALPHA(OUTL,FRT,FRT)
        DELTAMU =-PMU*MU(OUTL,FRT,FRT)
      ENDIF
C IF DELTA AND DLETA AVERAGE EQUALS ZERO
      IF (DELTAV*DELTA(OUTL,FRT,FRT) .EQ. 0.0)THEN
        DELTAP=0.0
        DELTAMU=0.0
      ENDIF
      DELTA(OUTL,FRT,FRT)=DI1
      ALPHA(OUTL,FRT,FRT)=ALPHA(OUTL,FRT,FRT)+DELTAP
C SET UPPER BOUND TO ALPHA
C IF COMPUTED ALPHA IS LARGER THAN UPPER BOUND
C THEN SET ALPHA TO UPPER BOUND
      IF(ALPHA(OUTL,FRT,FRT) .GT. ALPHAMAX)THEN
        ALPHA(OUTL,FRT,FRT)=ALPHAMAX
      ENDIF
      MU(OUTL,FRT,FRT)=MU(OUTL,FRT,FRT)+DELTAMU
C     GIVE THE BOUND OF MU
      IF(MU(OUTL,FRT,FRT) .GT. MUMAX)THEN
        MU(OUTL,FRT,FRT)=MUMAX
```

```
      ENDIF
      DW(LAYER,INL,FRT)=ALPHA(OUTL,FRT,FRT)*DI1
     * +MU(OUTL,FRT,FRT)*DW(LAYER,INL,FRT)
C      IF(DW(LAYER,INL,FRT) .GT. DWMAX)THEN
C        DW(LAYER,INL,FRT)=DWMAX
C      ENDIF
     DO 500 I=1,NMNODE(LAYER-1)
      DI1=DI*NODEOUT(LAYER-1,I)
C COMPUTE DELTA AND DELTA AVERAGE
      DELTAV=(1.0-THETA)*DI1+THETA*DELTA(OUTL,FRT,I)
C IF DELATA AND DELTA AVERAGE HAVE SAME SIGNS
      IF(DELTAV*DELTA(OUTL,FRT,I+1) .GT. 0.0)THEN
        DELTAP=KALPHA*EXP(-GALPHA * ABS(DELTAV))
        DELTAMU=KMU*EXP(-GMU * ABS(DELTAV))
      ENDIF
      IF(DELTAV*DELTA(OUTL,FRT,I+1) .LT. 0.0)THEN
C IF DELTA AND DELTA AVERAGE HAV OPPOSITE SIGNS
        DELTAP=-PALPHA*ALPHA(OUTL,FRT,I+1)
        DELTAMU=-PMU*MU(OUTL,FRT,I+1)
      ENDIF
C IF DELTA OR DELTA AVERAGE EQUALS ZERO
      IF(DELTAV*DELTA(OUTL,FRT,I+1) .EQ. 0.0)THEN
        DELTAP=0.0
        DELTAMU=0.0
      ENDIF
      DELTA(OUTL,FRT,I+1)=DI1
      ALPHA(OUTL,FRT,I+1)=ALPHA(3,1,I+1)+DELTAP
C SET UPPER BOUND TO ALPHA. IF ALPHA COMPUTED IS LARGER
C THAN UPPER BOUND THEN SET IT TO UPPER BOUND
      IF(ALPHA(OUTL,FRT,I+1) .GT. ALPHAMAX)THEN
        ALPHA(OUTL,FRT,I+1)=ALPHAMAX
      ENDIF
      MU(OUTL,FRT,I+1)=MU(OUTL,FRT,I+1)+DELTAMU
C      GIVE THE BOUND OF MU
      IF(MU(OUTL,FRT,I+1) .GT. MUMAX)THEN
        MU(OUTL,FRT,I+1)=MUMAX
```

```
      ENDIF
      DW(OUTL,FRT,I+1)=ALPHA(OUTL,FRT,I+1)*DI1
     * + MU(OUTL,FRT,I+1) * DW(OUTL,FRT,I+1)
C      IF(DW(OUTL,FRT,I+1).GT. DWMAX)THEN
C        DW(OUTL,FRT,I+1)=DWMAX
C      ENDIF
 500  CONTINUE
C
C THE FOLLWING IS THE SAME AS ABOVE EXCEPT FOR HIDDEN LAYER
     DO 510 I=1,NMNODE(LAYER-1)
      DI=DI*W(OUTL,FRT,I+1) * FUND(NODEIN(HHDL,I))
      DI1=DI*FUN(W(HHDL,I,1))
C CALCULATE ALPHA AND MU
C COMPUTE DELTA AND DELTA AVERAGE
      DELTAV=(1.0-THETA)*DI1+THETA*DI1
      IF(DELTAV*DELTA(HHDL,I,FRT) .GT. 0.0)THEN
        DELTAP=KALPHA*EXP(-GALPHA*ABS(DI1))
        DELTAMU=KMU*EXP(-GMU*ABS(DI1))
      ENDIF
      IF(DELTAV * DELTA(HHDL,I,FRT) .LT. 0.0)THEN
        DELTAP=-PALPHA*ALPHA(HHDL,I,FRT)
        DELTAMU=-PMU*MU(HHDL,I,FRT)
      ENDIF
      IF(DELTAV*DELTA(HHDL,I,FRT) .EQ. 0.0)THEN
        DELTAP=0.0
        DELTAMU=0.0
      ENDIF
      DELTA(HHDL,I,FRT)=DI1
      ALPHA(HHDL,I,FRT)=ALPHA(HHDL,I,FRT)+DELTAP
C      GIVE THE UPPER BOUND OF ALPHA
C IF ALPHA IS LARGER THAN UPPER BOUND
C THEN SET IT TO UPPER BOUND
      IF(ALPHA(HHDL,I,FRT) .GT. ALPHAMAX)THEN
        ALPHA(HHDL,I,FRT)=ALPHAMAX
      ENDIF
      MU(HHDL,I,FRT)=MU(HHDL,I,FRT)+DELTAMU
```

```
C      GIVE THE UPPER BOUND OF MU
       MU(HHDL,I,FRT)=MU(HHDL,I,FRT)+DELTAMU
       IF(MU(HHDL,I,FRT) .GT. MUMAX)THEN
         MU(HHDL,I,FRT)=MUMAX
       ENDIF
       DW(HHDL,I,FRT)=ALPHA(HHDL,I,FRT)*DI1+MU(HHDL,I,FRT)
       DO 520 J=1,NMNODE(LAYER-2)
         DI1=DI*NODEOUT(LAYER-2,J)
C COMPUTE DELTA AND DELTA AVERAGE
         DELTAV=(1.0-THETA)*DI1+THETA*DI1
         IF(DELTAV*DELTA(HHDL,I,J+1) .GT. 0.0)THEN
           DELTAP=KALPHA*EXP(-GALPHA*ABS(DI1))
           DELTAMU=KMU*EXP(-GMU*ABS(DI1))
         ENDIF
         IF(DELTAV*DELTA(HHDL,I,J+1) .LT. 0.0)THEN
            DELTAP=-PALPHA*ALPHA(HHDL,I,J+1)
            DELTAMU=-PMU*MU(HHDL,I,J+1)
         ENDIF
         IF(DELTAV*DELTA(HHDL,I,J+1) .EQ. 0.0)THEN
           DELTAP=0.0
           DELTAMU=0.0
         ENDIF
         DELTA(HHDL,I,J+1)=DI1
         ALPHA(HHDL,I,J+1)=ALPHA(HHDL,I,J+1) + DELTAP
C        GIVE THE UPPER BOUND OF ALPHA
         IF(ALPHA(HHDL,I,J+1) .GT. ALPHAMAX)THEN
           ALPHA(HHDL,I,J+1)=ALPHAMAX
         ENDIF
         MU(HHDL,I,J+1)=MU(HHDL,I,J+1)+DELTAMU
C        GIVE THE UPPER BOUND OF MU
         MU(HHDL,I,J+1)=MU(HHDL,I,J+1)+DELTAMU
         IF(MU(HHDL,I,J+1) .GT. MUMAX)THEN
           MU(HHDL,I,J+1)=MUMAX
         ENDIF
         DW(HHDL,I,J+1)=ALPHA(HHDL,I,J+1)*DI1+MU(HHDL,I,J+1)
     *   * DW(HHDL,I,J+1)
```

```
C         IF(DW(HHDL,I,J+1) .GT. DWMAX)THEN
C           DW(HHDL,I,J+1)=DWMAX
C         ENDIF
 520    CONTINUE
 510  CONTINUE
      END
C
C
C*****************************************************************
      FUNCTION FUND(X)
C DERIVATIVE OF TRANSFER FUNCTION
C COMPUTE DERIVATIVE OF TRANSFER FUNCTION
C BY TRANSFUNCTION
C INPUT: X
C OUTPUT: FUND
      REAL FUND,X,Y
C
      FUND=(1.0+FUN(X))*(1.0-FUN(X))
      Y=FUND
      RETURN
      END
C
C
C*****************************************************************
      FUNCTION FUN(X)
C TRANSFER FUNCTION OF SIGMOID
C INPUT:X
C OUTPUT: FUN
      REAL FUN, X
C
      FUN=(EXP(X)-EXP(-X))/(EXP(X)+EXP(-X))
      RETURN
      END
C
C
C*****************************************************************
```

```
      SUBROUTINE RDINPUT(T,N,M,FLAG,NDPT,LST)
C READ TRAINING DATA
C FROM TRAINING DATA FILE WHICH IS NOT
C NORMALIZED
C OUTPUT : T(N,M)
      REAL T(N,M)
      INTEGER N,M,NDPT,LST
      INTEGER I,J,IN
C
      IN=2
      IF(FLAG .EQ. 1)THEN
       OPEN(IN,FILE='ntrain.dat')
      ELSE
        OPEN(IN, FILE='ntest.dat')
      ENDIF
      DO 600 I=1,NDPT
        READ(4,610) (T(I,J),J=1,LST-1),T(I,LST)
600   CONTINUE
610   FORMAT(24F6.2,F9.6)
      CLOSE(IN)
      RETURN
      END
C
C
C*******************************************************
      SUBROUTINE RANWT(W,NMNODE,LAYER,N,M,INL,HHDL)
C INTIALIZE NEURAL NETWORK BY RANDOMNIZE ITS WEIGHTS
      REAL W(LAYER,N,M)
      INTEGER NMNODE(N),LAYER,N,M,I,J
C
      DO I=1,NMNODE(HHDL)+1
        W(LAYER,FRT,I)=RAND()
      ENDDO
      DO 700 I=1,NMNODE(HHDL)
        DO 710 J=1, NMNODE(INL)+1
710     CONTINUE
```

```
 700 CONTINUE
    END
C
C
C***********************************************************
    FUNCTION FINDMAX(T,N,M,II,NDPT)
C FIND MINNIMUM AND MAXIMUM OF EACH FIELD OF
C WHOLE TRAINING SET
C INPUT:TEMPERATURE T(N,M)
C OUTPUT: MIN(N), MAX(N)
    REAL FINDMAX,T(N,M)
    INTEGER N,M,II,I

    FINDMAX=0.0
    DO I=1,NDPT
      IF(T(I,II) .GT. FINDMAX)THEN
        FINDMAX=T(I,II)
      ENDIF
    ENDDO
    RETURN
    END
C
C
C***********************************************************
    FUNCTION FINDMIN(T,N,M,II,NDPT)
C FIND MINIMUM ELEMENT FROM T
    REAL FINDMIN,T(N,M)
    INTEGER N,M,II,I

C SET INITIAL VALUE FOR COMPARISON
    FINDMIN = 1000.0
    DO I=1,NDPT
      IF(T(I,II) .LT. FINDMIN)THEN
        FINDMIN=T(I,II)
      ENDIF
    ENDDO
```

```
      RETURN
      END
C
C
C************************************************************
      SUBROUTINE MNTAB(T,MIN,MAX,N,M,LST)
C SET UP MIN-MAX TABLE
      REAL T(N,M),MIN(LST),MAX(LST)
      INTEGER N,M,I,LST
C
      DO I=1,LST
        MAX(I)=FINDMAX(T,N,M,I)
        MIN(I)=FINDMIN(T,N,M,I)
      END DO
      END
C
C
C************************************************************
      SUBROUTINE INITDW(DW,DELTA,ALPHA,LAYER,N,M)
C   INITIALIZE DELTA WEIGHT AND PARAMETERS
C
      REAL DW(LAYER,N,M),DELTA(LAYER,N,M),ALPHA(LAYER,N,M)
C
      INTEGER N,M,NMNODE(LAYER),LAYER
C
      INTEGER I,J
      REAL AX,DX
C
C   SET DELTA INITIAL VALUE
      DX=0.1
C   SET ALHPA  INITIAL VALUE
      AX=0.1
      DO I=1,N
        DW(LAYER,FRT,I)=RAND()
        DELTA(LAYER,FRT,I)=AX
        ALPHA(LAYER,FRT,I)=DX
```

```
      END DO
      DO 800 I=1,N
        DO 810 J=1,M
          DW(HHDL,I,J)=RAND()
          DELTA(HHDL,I,J)=DX
          ALPHA(HHDL,I,J)=AX
 810    CONTINUE
 800  CONTINUE
      END
C
C
C***********************************************************
C THIS IS THE MAIN PROGRAM FOR STEEPEST DESCENT IN BATCH
C MODE WITH LINE SEARCH. ALL OF SUBROUTINE OR FUNCTION CALLS
ENCOUNTEREDC IN THIS PORTION IS EXACTLY THE SAME AS IN THE PROGRAM
C ABOVE AND IS NOT LISTED HERE FOR  AVOIDING REPEATING.
C IT CAN BE EXECUTED INDEPENDENTLY. DOING SO IS ONLY
C FOR THE REASON OF PROGRAMMING CONVENIECE.


      INTEGER I,J,II
      REAL MOISTURE,EPSILON,SUMERR
     *,NODEIN(30,30),NODEOUT(30,30),W(3,30,30),DW(3,30,30)
     *,DELTA(3,30,30),ALPHA(3,30,30),DWA(3,30,30),TEMP(310,26)
     *,MIN(25),MAX(25),RI,WM
     *,TW(3,30,30),RDFT,ENFT,LCOEF,FV,BR
     * ,HIGH,LOW,OFFSET,SCALE,YI,SI,YIO,SIO
      INTEGER NMNODE(3),TRMD,EPOCH,EPLENGTH,
     *NN,MM,EPCNT,JJ,INL,HHDL,OUTL
     *FLAG,MAXEP,MAXSCH
C
C INPUT LAYER NUMBER
      INL=1
C HIDDEN LAYER NUMBER
      HHDL=2
C OUTOUT LAYER NUMBER
```

```
      OUTL=3
C NUMBER OF NODES IN INPUT LAYER
      NMNODE(INL)=25
C NUMBER OF NODES IN HIDDEN LAYER
      NMNODE(HHDL)=4
C NUMBER OF NODES IN OUTPUT LAYER
      NMNODE(OUTL)=1
C NUMBER OF LAYERS
      LAYER=3
C NUMBER OF UNITS IN INPUT LAYER
      N=NUMNODE(INL)
C NUMBER OF UNITS IN HIDDEN LAYER
      M=NUMNODE(HHDL)
C CONVERGENCE CRITERION
      EPSILON = 0.037
C SET SUM SQUARE ERROR TO ZERO
      SUMERR = 0.0
C FLAG FOR READ TRAINING FILE OR TEST FILE
C FLAG=1--READ TRAINING FILE; FLAG=2--READ TESTING FILE
      FLAG=1
C THE NUMBER OF PRESENTATIONS BEFORE UPDATING WEIGHTS
      EPLENGTH = 154
C OUTPUT RANGE OF UPPER AND LOWER BOUND
      HIGH = 1.0
      LOW = 0.0
C NUMBER OF DATA POINTS IN TRAINING SET OR TESTING SET
      NDPT=154
C THE FIRST NODE IN ONE LAYER
      FRT=1
C THE LAST NODE IN INPUT LAYER
      LST=25
C THE COUNTER FOR UPDATE WEIGHTS
      EPOCH = 0
C MAXIMU VALUE OF WEIGHT
      WM=50.0
C DECREMENTING RANGE
```

```
      BR=0.005
C MAXIMUM NUMBER OF EPOCH OF TRAINING
      EPMAX=1000
C SET ENLARGE FACTOR
      ENFT=1.5
C SET REDUCE FACTOR
      RDF =0.75
C SET INITIAL LEARNING COEFICIENT
      LCOEF=0.2
C
C OUTPUT DEVICE
      OUT1=4
      OUT2=6
C
      MM = 0
C
      OPEN(OUT22,FILE='bp.dat')
C READ TRAINING DATA INTO BUFFER
      CALL RDINPUT(TEMP,NDPT,NMNODE(INL),FRT)
C RANDOMIZE CONNECTION WEIGHTS
      CALL  RANWEIGHT(W,NMNODE,LAYER,NMNODE(INL),NMNODE(HHDL))
C INPUT MINMAX TABLE FOR NORMALIZATION
      CALL MMTAB(TEMP,MIN,MAX,NDPT,NMNODE(INL))
C INITIALIZE DELTA WEIGHTS
      CALL INITDW(DW,DELTA)
C
 50   DO 52 KK=1,NDPT
C
C PRESENT ONE VECTER TO INPUT LAYER
        DO 20 J=1,NMNODE(INL)
          NODEOUT(INL,J)=TEMP(KK,J)
 20     CONTINUE
C

      SIO=NODEOUT(FRT,LST)
C
```

```
C NORMALIZE THE INPUT PATTERN
    DO I=1,NMNODE(INL)
      SCALE=(HIGH-LOW)/(MAX(I)-MIN(I))
      OFFSET=(MAX(I)*LOW-MIN(I)*HIGH)/(MAX(I)-MIN(I))
      NODEOUT(INL,I)=NODEOUT(INL,I)*SCALE+OFFSET
    ENDDO

C
C FORWARD INPUT VECTER TO OUTPUT LAYER
    CALL FDINPUT(W,NODEIN,NODEOUT,NMNODE,LAYER,N,M)
C
C RESCALE OUTPUT TO THE ORIGINAL RANGE
    SI=NODEOUT(FRT,LST)
    YI=NODEOUT(LAYER,FRT)
    SCALE=(MAX(LST)-MIN(LST))/(HIGH-LOW)
    OFFSET=(HIGH*MIN(LST)-LOW*MAX(LST))/(HIGH-LOW)
    YIO=YI*SCALE+OFFSET

C     WRITE(*,23) YIO, SIO
 23  FORMAT(1X,2F8.6)
C
C COMPUT THE SUM SQUARED ERROR
    SUMERR=SUMERR+(SIO-YIO)*(SIO-YIO)
C
C COMPUT DELTA WEIGHT
     CALL STDBP(W,DW,NODEIN,NODEOUT,NMNODE,LAYER,N,M,
    * SI,YI)
C ACCUMULATE DELTA WEIGHTS
C DO I=1,NMNODE(HHDL)
    DWA(LAYER,FRT,I)=DWA(LAYER,FRT,I)+DW(LAYER,FRT,I)
    ENDDO
     DO 25 I=1,NMNODE(HHDL)
       DO 28 J = 1,NMNODE(INL)
         DWA(HHDL,I,J)=DWA(HHDL,I,J)+DW(HHDL,I,J)
 28    CONTINUE
 25  CONTINUE
```

```
C
 52  CONTINUE
C
    FV=SUMERR
 15  SUMERR=SQRT(SUMERR/NDPT)
C OUTPUT NUMBER OF EPOCHS AND ROOT SQUARED SUM ERROR
    WRITE(OUT2,31) EPCNT,SUMERR
    WRITE(*,31) EPCNT,SUMERR
    EPCNT=EPCNT+1
 31  FORMAT(1X,I6,' ', F8.6)
C IF RMS LESS THAN EPSILON, THEN BEGIN TO TEST NETWORK
    IF(SUMERR .LT. EPSILON)THEN

C READ TEST DATA FROM FILE
      FLAG=2
 32    CALL RDINPUT(TEMP,NDPT,NMNODE(INL),FLAG)
      OPEN(OUT1,FILE='bp.nnr')
      WRITE(OUT1,45) COUNTER
      DO K=1,NDPT
        DO 21 J=1,NMNODE(INL)
          NODEOUT(INL,J)=TEMP(K,J)
          IF(NODEOUT(1,J) .EQ. 0.0 )THEN
            STOP
          ENDIF
 21       CONTINUE
        SIO=NODEOUT(FRT,LST)
C NORMALIZE THE INPUT VECTER
        DO 22 I=1,NMNODE(INL)
          SCALE=(HIGH-LOW)/(MAX(I)-MIN(I))
          OFFSET=(MAX(I)*LOW-MIN(I)*HIGH)
           /(MAX(I)-MIN(I))
          NODEOUT(INL,I)=NODEOUT(INL,I)*SCALE+OFFSET
 22       CONTINUE

C FORWARD THE INPUT VECTERS
        CALL FDINPUT(W,NODEIN,NODEOUT,NMNODE,LAYER,N,M)
```

```
        YI=NODEOUT(LAYER,FRT)
        SCALE=(MAX(LST)- MIN(LST))/(HIGH-LOW)
        OFFSET=(HIGH*MIN(LST)-LOW*MAX(LST))
              /(HIGH-LOW)
        YIO=YI*SCALE+OFFSET

        WRITE(OUT1,23) YIO,SIO

      END DO
      CLOSE(OUT1)
      CLOSE(OUT2)
      STOP
    ENDIF
C
    SUMERR=0
C
 59  DO I = 1,NMNODE(HHDL)+1
      TW(LAYER,FRT,I)=W(LAYER,FRT,I)+LCOEF*DWA(LAYER,FRT,I)
    ENDDO
    DO 35 I=1,NMNODE(HHDL)
      DO 38 J=1,NMNODE(1)
        TW(HHDL,I,J)=W(HHDL,I,J)+LCOEF*DWA(HHDL,I,J)
 38    CONTINUE
 35  CONTINUE
C
    SUMERR=0
C
C IF NUMBER OF EPOCHS LARGER THAN MAXIMU NUMBER
C THEN STOP TRAINING
    IF(EPCNT .EQ. MAXEP )THEN
      GOTO 32
    ENDIF
    DO 53 KK=1,NDPT

      DO  J=1,NMNODE(INL)
        NODEOUT(INL,J)=TEMP(KK,J)
```

```
      ENDDO
C
C FORWARD INPUT VECTER
    CALL FDINPUT(TW,NODEIN,NODEOUT,NMNODE,LAYER,N,M)
C
    SI=NODEOUT(FRT,LST)
    YI=NODEOUT(LAYER,FRT)
C
C COMPUTE SUM SQUARE ERROR
    SUMERR=SUMERR+(SI-YI)*(SI-YI)
 53  CONTINUE
C
C     WRITE(*,31) EPCNT,SUMERR
     IF((SUMERR-FV) .LE. BR )THEN
       WRITE(*,45) NN
       NN = 0
       DO I = 1,NMNODE(HHDL)+1
         W(LAYER,FRT,I)=W(LAYER,FRT,I)+LCOEF*DWA(LAYER,FRT,I)
C IF WEIGHT LARGER OR LESS THAN  BOUND
C THEN SET IT TO THE BOUND
         IF(W(LAYER,FRT,I) .GT. WM)THEN
           W(LAYER,FRT,I)=WM
         ENDIF
         IF(W(LAYER,FRT,I) .LT. -WM)THEN
           W(LAYER,FRT,I)=-WM
         ENDIF
       ENDDO
       DO 40 I=1,NMNODE(HHDL)
         DO 42 J=1,NMNODE(INL)
           W(HHDL,I,J)=W(HHDL,I,J)+LCOEF*DWA(HHDL,I,J)
C SET WEIGHT TO THE UPPER OR LOWER BOUND
C IF IT LARGER OR LESS THAN ITS BOUNDS
           IF(W(HHDL,I,J) .GT. WM )THEN
             W(HHDL,I,J)=WM
           ENDIF
           IF(W(HHDL,I,J) .LT. -WM)THEN
```

```
          W(HHDL,I,J)=-WM
        ENDIF
42      CONTINUE
40    CONTINUE
C IF SUCCESS THEN ENLARGE FACTOR
    LCOEF=ENFT*LCOEF
    GOTO 70
    ENDIF
C
C IF SEARCH FAILURE THEN REDUCE FACTOR
C AND CONTINUE TRY
    LCOEF=RDFT*LCOEF
    NN=NN+1
    SUMERR=0
    IF(NN .GE. MAXSCH)THEN
      NN = 0
      DO I=1,NMNODE(HHDL)+1
        W(LAYER,FRT,I)=W(LAYER,FRT,I)+LCOEF*DWA(LAYER,FRT,I)
C SET WEIGHT TO UPPER OR LOWER BOUND
C IF IT LARGER OR LESS THAN ITS BOUNDS
        IF(W(LAYER,FRT,I) .GT. WM)THEN
          W(LAYER,FRT,I)=WM
        ENDIF
        IF(W(LAYER,FRT,I) .LT. -WM)THEN
          W(LAYER,FRT,I)=-WM
        ENDIF
      END DO
C UPDATE WEIGHTS IF SUCCESS
      DO 41 I=1,NMNODE(HHDL)
        DO 43 J=1,NMNODE(INL)
          W(HHDL,I,J)=W(HHDL,I,J)+LCOEF*DWA(HHDL,I,J)
C SET WEIGHT TO UPPER OR LOWER BOUND
C IF IT LARGER OR LESS THAN ITS BOUNDS
          IF(W(HHDL,I,J) .GT. WM )THEN
            W(HHDL,I,J)=WM
          ENDIF
```

```
          IF(W(HHDL,I,J) .LT. -WM)THEN
            W(HHDL,I,J)=-WM
          ENDIF
43        CONTINUE
41      CONTINUE
        GOTO 70
      ENDIF
      GOTO 59
C
C     WRITE(*,29) LCOEF
 29   FORMAT(1X,F8.6)
      SUMERR=0.0
C
C RESET DELTA WEIGHTS
 70   DO I=1,NMNODE(HHDL)
        DWA(LAYER,FRT,I)=0.0
      END DO
      DO 26 I=1,NMNODE(HHDL)
        DO 27 J=1,NMNODE(INL)
          DWA(HHDL,I,J)=0.0
 27     CONTINUE
 26   CONTINUE
 45   FORMAT(1X,I5)
      SUMERR=0.0
      GOTO 50
 60   END
```

VITA

Xiaoji Liu

Candidate for the Degree of

Master of Science

Thesis: A COMPARISON STUDY OF FEEDFORWARD FULLY-CONNECTED NEURAL NETWORKS VS. CASCADE CORRELATION NETWORKS FOR PREDICTION OF SOIL MOISTURE CONTENT

Major Field: Computer Science

Biographical:

Personal Data: Born in Taizhou, Jiangsu Province, P. R. China, July 1957, the son of Huilin Wang and Junhua Liu.

Education: Graduated From First High School of Taizhou, Taizhou, Jiangsu Province, P. R. China in July 1975; received Bachelor of Science Degree in Applied Mathematics from East China Institute of Technology in January 1982; completed requirements for the Master of Science degree at Oklahoma State University in December 1994.

Professional Experience: Software engineer, North Industry Corporation of China, 1989 through 1992. Visiting engineer, IIT (Integrated Information Technology, Inc.), Santa Clara, California, from November 1988 to August 1989. Software engineer, North Industry Corporation of China, from January 1982 to July 1988.

Membership: China Computer Society; American Mathematical Society.