LAYER-WISE TRAINING OF FEEDFORWARD NEURAL NETWORKS BASED ON LINEARIZATION AND SELECTIVE DATA PROCESSING

By

Shawn David Hunt

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1992

ABSTRACT

LAYER-WISE TRAINING OF FEEDFORWARD NEURAL NETWORKS BASED ON LINEARIZATION AND SELECTIVE DATA PROCESSING

By

Shawn David Hunt

A class of algorithms is presented for training nonlinear feedforward neural networks using purely "linear" techniques. The algorithms are based upon linearizations of the network using error surface analysis, followed by a contemporary recursive least squares identification procedure which can be implemented using parallel processing. Specific algorithms are presented to estimate weights node-wise, layer-wise, and for estimating the entire set of network weights simultaneously. A procedure for modifying the algorithms to selectively use the training data and increase speed is also presented. A computationally inexpensive measure is developed with which to assess the effect of a particular training pattern on the weight estimates prior to its inclusion in any iteration. Data which do not significantly change the weights are not used in that iteration, obviating the computational expense of updating. Several experimental studies are presented showing the advantages of this class of algorithms. Specifically, the layer-wise algorithm is shown to be vastly superior to back-propagation in terms of the number of convergences and convergence rate. Additionally this algorithm is shown to be insensitive to the choice of initial weights and forgetting factor, eliminating two of the greatest problems in the implementation of existing training algorithms.

To my parents

David and Suzanne Hunt

Acknowledgments

I would like to thank John R. Deller, Jr., for the immense help in completing this degree. The biggest help was probably learning how (even if I do not necessarily follow it) to write a technical paper. I have heard that a Ph.D. degree means you can do research. Of course, if you cannot transmit this to others in a clear, concise manner, it is of little use. I also must give thanks for his defense (during my defense, especially the proposal), and his willingness to see me and listen to any problem. One of the greatest benefits of being at Michigan State was being in a lab where the students not only got along, but where there was a spirit of cooperation and friendliness. For this I must thank both Dr. Deller and Joan (for keeping El Estimado happy).

I would also like to thank the members of my committee, Dr. Majid Nayeri, Dr. Norman Birge, and Dr. Fathi Salam. It was an unexpected surprise to find out that the committee members had not only read the complete dissertation, but in great detail. I could not have asked for more.

Money! For this I must thank the University of Puerto Rico for granting me a fellowship. This work was also supported by National Science Foundation under Grant No. MIP-9016734, and the Office of Naval Research under Contract No. N00014-91-J-1329.

I also wish to thank the members of the East Lansing Baha'i community for the support which has made these years truly enjoyable. No, not only enjoyable, great! First the Nazerians, Keyvan, Farzaneh, Mike, Alan and Lily for all the great food, the thanksgivings, the borrowed cars, and of course the tennis. Njeru Murage, Roya

Mavaddat, Bruce Johnson and Irene Shim (a token Baha'i), for the most enlightening discussions on every imaginable topic. (Mike, Alan and I also had interesting discussions but these were enlightening on a different level.) Also the Smiths, David, Melanie, Amber, Amelia, and Zack(God is One), for the incredible spirit at the sunday night firesides. Joe and Judy DeFlorio! (and of course Sophia) Could we have done it without you? Dominic, for being the only five year-old to be interested in hearing me talk about black holes. Also thanks to Franco and Shahnaz Damazio for being a little persian and worrying. Also to the members who enlightened the club here on campus over the years, Gail Etzenhauser(lets go!), Hami Missagieh(Haaaaami), Ohmead Bashirelahi(Sasha who?), Farnoush(why sleep) and Kianoush(let go to Rezas) Sinai, Tahirih Steen, Rick and Eve Martin, Mark Knox(lets teach), Gill Munro(caderas), Greg(el tigre) and Valerie(hi babe) Heikes, and John Homan(thanks for the music). I know I have left out many, please forgive me.

Anne and Alicia, thank you both for just being there and thank you for the prayers.

Contents

List of Tables viii

List of Figures ix

1 Introduction and Background 1
- 1.1 Introduction . . . 1
  - 1.1.1 The Promise of Neural Networks . . . 1
  - 1.1.2 Notation and Objectives . . . 3
- 1.2 The Feedforward Neural Network (FNN) . . . 5
- 1.3 Training Algorithms for FNNs . . . 9
  - 1.3.1 Performance Measures . . . 9
  - 1.3.2 Training Efficiency . . . 13
  - 1.3.3 The Back-Propagation Algorithm . . . 14
  - 1.3.4 Techniques for Improving Convergence . . . 15
  - 1.3.5 Complexity Measures . . . 19

2 New Training Algorithms 20
- 2.1 Introduction . . . 20
- 2.2 Linearized Training Algorithms . . . 20
  - 2.2.1 Node-wise Weight Updating . . . 21
    - 2.2.1.1 Two Layer Network . . . 21
    - 2.2.1.2 L Layer Network . . . 28
  - 2.2.2 Layer-wise Weight Updating . . . 30
    - 2.2.2.1 Two Layer Network . . . 30
    - 2.2.2.2 L Layer Network . . . 35
  - 2.2.3 Network-wise Weight Updating . . . 38
    - 2.2.3.1 Two Layer Network . . . 38
    - 2.2.3.2 L Layer Network . . . 38
- 2.3 Solving the Linearized Equations . . . 40
  - 2.3.1 Solution by MIL-WRLS . . . 40
  - 2.3.2 Solution by QR-WRLS . . . 41

3 Data Reduction Algorithm 46
- 3.1 Introduction . . . 46
- 3.2 Data Reduction Algorithm . . . 46

4 Simulation Results **59**
4.1 Introduction 59
4.2 Implementation Studies 60
4.3 Layer Updating and Network Updating 66
4.3.1 Initialization 66
4.3.2 Forgetting Factor 70
4.3.3 Small Training Sets 77
4.3.4 Large Training Set 85
4.4 Simulations of the Data Reduction Algorithm 93

5 Conclusions and Future Research **102**
5.1 Algorithmic Developments 102
5.1.1 Training Speed 102
5.1.2 Layer-wise Training Algorithm 103
5.1.3 QR-WRLS 103
5.1.4 Data Reduction Algorithm 104
5.2 Future Research 104

Appendix **109**

List of Tables

1.1 Complexities per iteration and normalized complexities of FNN training algorithms. M is the number of nodes per layer. L is the number of layers. 19
4.2 Number of convergences per 100 sets of initial weights 64
4.3 Convergence results for various initial weights, 2-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence. 68
4.4 Convergence results for various initial weights, 4-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence. 68
4.5 Convergence results for various forgetting factors, 2-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence. 78
4.6 Convergence results for various forgetting factors, 4-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence. 80
4.7 Convergence results for various forgetting factors, 4-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence. 82
4.8 Iterations to convergence of the BP algorithm for various training sets and varying number of hidden layer nodes. 90
4.9 Iterations to convergence of the node-wise algorithm for various training sets and varying number of hidden layer nodes. 91
4.10 Iterations to convergence of the layer-wise algorithm for various training sets and varying number of hidden layer nodes. 93
4.11 Iterations to convergence of the network-wise algorithm for various training sets and varying number of hidden layer nodes. 94
4.12 Number of correctly classified training patterns for the large training set for varying number of inner layer nodes. 94

List of Figures

1.1 A single node. 6
1.2 Example of a sigmoid function. 7
1.3 A two-layer network. 8
2.4 Weight estimation using recursive QR-WRLS 43
4.5 Network architectures used in the simulation studies. 1. the 2-bit parity checker, 2. the 4-bit parity checker, and 3. the 4-bit bit counter. 62
4.6 Average error in dB for the X-OR implementations vs. iteration number. 1. back-propagation; 2. A-S algorithm; 3. QR-WRLS. 65
4.7 Average error in dB for the BP implementation of the 2-bit parity checker vs. iteration number using different methods of selecting initial weights. 69
4.8 Average error in dB for the node-wise implementation of the 2-bit parity checker vs. iteration number, using different methods of selecting initial weights. 69
4.9 Average error in dB for the layer-wise implementation of the 2-bit parity checker vs. iteration number, using different methods of selecting initial weights. 70
4.10 Average error in dB for the network-wise implementation of the 2-bit parity checker vs. iteration number, using different methods of selecting initial weights. 71
4.11 Average error in dB for the BP implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights. 72
4.12 Average error in dB for the node-wise implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights. 72
4.13 Average error in dB for the layer-wise implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights. 73
4.14 Average error in dB for the network-wise implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights. 74

4.15 Average error in dB for the QR-WRLS X-OR implementation vs. iteration number, using different forgetting factors and weight change constraints. 1. $\nu = 0.98, \gamma = 0.2$; 2. $\nu = 0.98, \gamma = 1.0$; 3. $\nu = 0.1, \gamma = 1.0$; 4. $\nu = 0.1, \gamma = 0.2$; where ν is the forgetting factor and γ is the weight constraint. 75
4.16 Average error in dB for the node-wise implementation of the 2-bit parity checker vs. iteration number, using different forgetting factors. 78
4.17 Average error in dB for the layer-wise implementation of the 2-bit parity checker vs. iteration number, using different forgetting factors. 79
4.18 Average error in dB for the network-wise implementation of the 2-bit parity checker vs. iteration number, using different forgetting factors. 79
4.19 Average error in dB for the node-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors. 80
4.20 Average error in dB for the layer-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors. 81
4.21 Average error in dB for the network-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors. 81
4.22 Average error in dB for the node-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors. 82
4.23 Average error in dB for the layer-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors. 83
4.24 Average error in dB for the network-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors. 83
4.25 Training set A. 85
4.26 Training set C. 86
4.27 Training set E. 86
4.28 Training set G. 87
4.29 Training set I. 87
4.30 Training set K. 88
4.31 Training set L. 88
4.32 Training set M. 89
4.33 Training set N. 89
4.34 Large training set. 91
4.35 Average error vs. iteration number for the BP implementation, training with the large training set. 1. 6 nodes; 2. 7 nodes; 3. 8 nodes. . . 92
4.36 Average error vs. iteration number for the Node-wise implementation, training with the large training set. 1. 6 nodes; 2. 7 nodes; 3. 8 nodes. 92
4.37 Average error vs. iteration number for the Layer-wise implementation, training with the large training set. 1. 6 nodes; 2. 7 nodes; 3. 8 nodes. 95
4.38 Average error vs. iteration number for the data reduction algorithm using 6 hidden layer nodes, training with the large training set. . . . 96
4.39 Correctly classified training patterns vs. iteration number for the data reduction algorithm using 6 hidden layer nodes and the large training set. 97

4.40 Average error vs. iteration number for the data reduction algorithm using 7 hidden layer nodes, training with the large training set. . . . 97
4.41 Correctly classified training patterns vs. iteration number for the data reduction algorithm using 7 hidden layer nodes and the large training set. . . . 98
4.42 Average error vs. iteration number for the data reduction algorithm using 8 hidden layer nodes, training with the large training set. . . . 98
4.43 Correctly classified training patterns vs. iteration number for the data reduction algorithm using 8 hidden layer nodes and the large training set. . . . 99
4.44 Number of training patterns used vs. iteration number for the data reduction algorithm and a dividing factor of 30. 1. 6 nodes 2. 7 nodes 3. 8 nodes. . . . 99
4.45 Number of training patterns used vs. iteration number for the data reduction algorithm and a dividing factor of 100. 1. 6 nodes 2. 7 nodes 3. 8 nodes. . . . 100

Chapter 1

Introduction and Background

1.1 Introduction

1.1.1 The Promise of Neural Networks

This research is concerned with supervised training of Feedforward Artificial Neural Networks (FNNs). FNNs are a special case of a class of non-linear networks called Artificial Neural Networks (ANNs). ANNs have gained popularity in recent years due to their promise of solving previously intractable problems. ANNs have been applied to many problems in the areas of pattern recognition, pattern classification, and associative memories. Some of these problems are from the areas of speech recognition, computer vision, medical diagnosis and non-linear control. Included are such specific problems such as teaching a machine to read out loud or to play games. The usefulness of a computer that understands speech or can "see" has been known for decades. For this reason, research to solve problems in the areas mentioned above has been vigorous. However, despite all the research, solutions for many of these problems have not been found using conventional linear techniques such as linear regression, and non-linear techniques such as artificial intelligence implemented on a standard sequential computer. For this reason, it has been suggested that the difficulty in finding solutions to the problem does not lie in the fact that the correct algorithm

for implementation on a sequential computer has not been found, but rather that sequential computers are inherently unsuited to the problem. An architecture that does seem well-suited to the problem is that of organic neural networks, the brains of animals. Consequently, some properties from these organic neural networks have been emulated in the development of ANNs. Some of these properties are high interconnectivity, massive parallelism, and individual units that are non-linear. A major area of research is now ascertaining whether training algorithms for ANNs, algorithms designed to make the ANN solve the problems mentioned above, actually exist.

A number of recently published simulation results provide evidence that the FNN can solve useful problems, even though an explicit mathematical proof of how the solution was reached may not be possible at this time. One of the most widely referenced experiments is that of NETTALK [1]. In this experiment, a FNN was trained to read out loud. The input to the FNN was standard text, and the output was a phonetic alphabet. This phonetic alphabet was fed into a separate computer which allowed the output to be heard aloud. As impressive as it was to have a network read out loud, even more exciting is the description by the experimenters of how the learning was accomplished. It is reported that the network started off babling, slowly progressing, making mistakes reminiscent of a child learning how to read. This gave great encouragement that the properties borrowed from nature in the making of FNNs were indeed useful in approximating a real neural system. Another experiment, possibly even more impressive from the engineering point of view, was that of Nguyen and Widrow [2]. The experiment succeeded in teaching a FNN how to back up a semi-trailer truck simulated on a computer. This simulation result is significant because at present we cannot design a controller for this nonlinear control problem. This experiment clearly shows the great advantage of FNNs over other methods. In a typical system at present, the designer must specify all the variables. This is not the case with FNNs. For example, in any pattern recognition problem, the features

to be detected and used for selection must be determined by the designer. With a multilayer FNN, however, it is the system itself which determines what features to use. This can be of great advantage as shown in the previous experiment. We would, of course, like to be able to analyze mathematically all the properties of the FNN. Until then, however, it will still be of use.

1.1.2 Notation and Objectives

As this dissertation relies heavily on the solution of a set of linear equations a short discussion on notation is appropriate. Suppose we have a typical overdetermined system of linear equations. There are many methods for solving this system for the weights that give the least mean square error. Many popular methods are recursive, and are typically called *recursive least squares (RLS)*. A weighting factor can be easily included in the recursions to weigh each of the linear equations differently in finding the final solution. This we will call *weighted recursive least squares (WRLS)*. One method frequently used [5] is based on the matrix inversion lemma [5] and will be called MIL-WRLS here. A more contemporary recursive approach based on QR decomposition [3],[4] will be used in this dissertation and will be called QR-WRLS.

The research described in this dissertation is concerned with the method and speed of training FNNs. The main contributions have been:

1. To show the advantages of implementing linearized FNN training algorithms with QR-WRLS instead of the conventional MIL-WRLS algorithm. This is done by modifying one such algorithm, the Azimi-Sadjadi *et al.* (A-S) [6] algorithm to employ QR-WRLS and comparing a variety of simulation results to the A-S algorithm implemented with MIL-WRLS.

2. To develop a "linearized" training procedure that is faster than the ones reported in literature. This involves using conventional least mean square equa-

tions with new "linearized" inputs and outputs determined from the FNN, and searching for a way to determine inner layer target values faster than the back-propagation technique now widely used.

3. To apply matrix perturbation theory to the linearized neural networks in an attempt to increase convergence speed. This includes derivation of equations that are well-suited for application to the FNN and are also simple enough so that a computational saving is possible.

The first two objectives are addressed in Chapter 2. A number of training algorithms currently proposed for training FNNs use linearizations. These linearizations transform the training problem to one of solving a system of linear equations. This allows the implementation of these training algorithms by any algorithm for solving the least squared error problem in linear systems. Existing training algorithms use the conventional MIL-WRLS algorithm for implementation. This algorithm takes the form of two recursions. The recursive equations are numerically unstable [7] and require initial conditions which are generally unknown. Also, in training the FNN, a forgetting factor [7] must be included. Implementation with MIL-WRLS limits the types of forgetting factors that can be used. All of these problems are dealt with effectively using QR-WRLS. First, QR-WRLS has been shown to be more efficient than the MIL-WRLS algorithm [4]. It is also stable numerically. QR-WRLS also allows different forgetting methods to be used. One of the proposed training algorithms that uses MIL-WRLS is implemented with QR-WRLS . Simulation results showed a marked increase in the algorithm performance. This result should be very beneficial in stimulating the use of QR-WRLS for other training algorithms implemented using MIL-WRLS. A new linearization training algorithm is also introduced in Chapter 2.

In order to further increase the rate of convergence, matrix perturbation theory has been used. Inputs to the network along with the desired response – together called training patterns – are used to train the FNN. In general, the network does

not usually "learn" to correctly classify the training patterns the first time they are used. Neither is there any guarantee that the network will *ever* learn to classify all the training patterns correctly. However, in the training process the network usually correctly classifies more and more of the training patterns as they are used repeatedly. For this reason the training patterns are used many times. If there were a way to avoid using all the training patterns without changing the solution, training could be made more efficient. One method for doing this data reduction is presented in Chapter 3.

Simulation studies of the performance of QR-WRLS, of the new training algorithm, and of the data reduction algorithm are presented in Chapter 4.

Chapter 5 contains the conclusions and suggests areas of future research.

1.2 The Feedforward Neural Network (FNN)

The basic units of the FNN network are "neurons" or nodes. The model of the node to be used in the research involves a weighted sum of the inputs, and an output that is a non-linear function of this weighted input. Thus,

$$u = \sum_{j=1}^{N} w_j x_j \tag{1.1}$$

where u is the weighted input, w_j is the weight for the j^{th} input, and x_j is the j^{th} input. The output of the node is:

$$y = S(u) \tag{1.2}$$

where y is the output of the node, and $S(\cdot)$ is a nonlinear mapping. A representation is shown in Figure 1.1. The non-linearity that will be used in this research is the sigmoid function

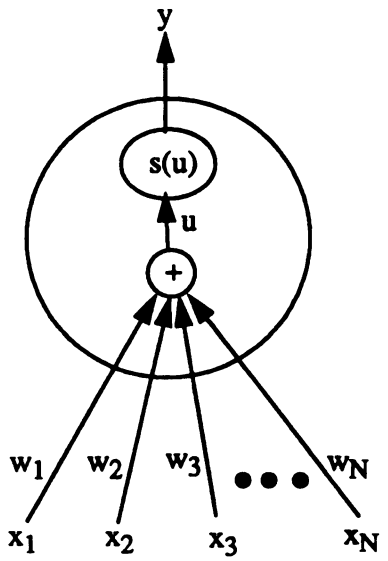

Figure 1.1: A single node.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1.3}$$

$S(\cdot)$ has the following properties: it is continuous, has continuous derivatives of all orders, is strictly increasing, and has finite limits as the argument goes to $+\infty$ or $-\infty$. A graph of the function is shown in Figure 1.2.

The nodes are grouped into layers. For clarity, in this dissertation a two-layer network is considered in the initial theoretical discussions[1], an example of which is illustrated in Figure 1.3. The generalization of the methods to an arbitrary number of layers follows the initial developments with two layers. Each node above the input layer in the FNN passes the sum of its weighted inputs through a non-linearity as in (1.2). The inputs to layer zero are external. The outputs of the last layer are the

[1]Some authors might choose to call this a *three* layer network. The bottom layer of "nodes" shall be designated as "layer zero" and not count it in the total number of layers. Layer zero is a set of linear nodes which simply pass the inputs unaltered. For this reason, circular nodes are not shown in the diagram.

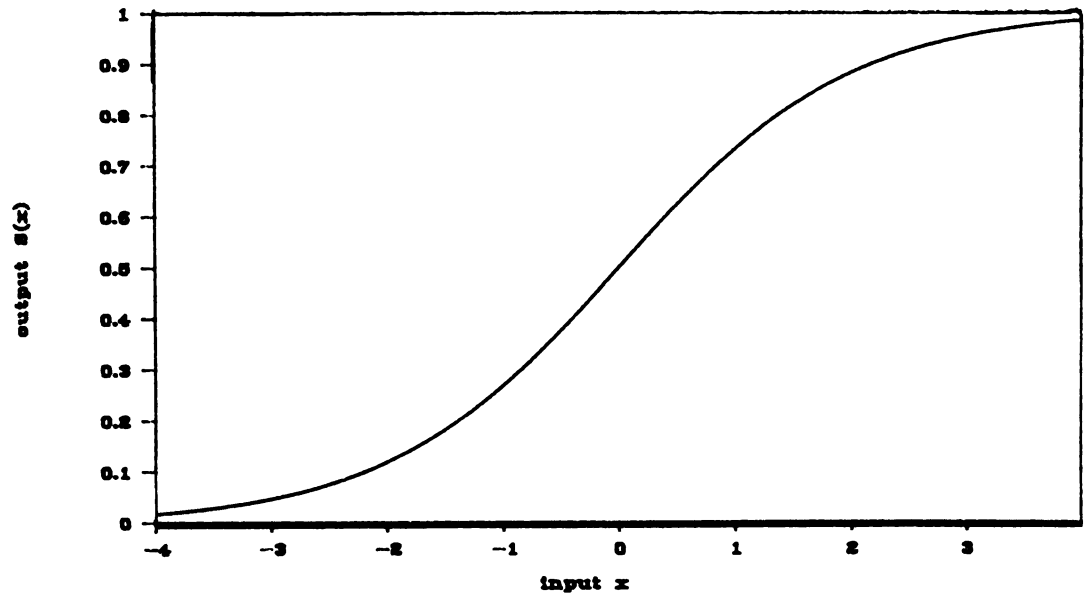

Figure 1.2: Example of a sigmoid function.

outputs of the network.

Let us now formalize the network and define notation. The number of nodes in layer i is denoted N_i, with N_0 indicating the number of input nodes at the bottom (input) of the network. The weights connecting to node k (k') of layer two (one) are held in the N_1-vector (N_0-vector) $\boldsymbol{w}_k = [w_{k,1}, \ldots, w_{k,N_1}]^T$ ($\boldsymbol{w}'_{k'} = [w'_{k',1}, \ldots, w'_{k',N_1}]^T$). The inputs to the nodes in all layers except the first are the outputs of the layer below. We denote by N the number of *training patterns*

$$\left\{\left(\boldsymbol{x}(n) = [x_1(n), x_2(n), ..., x_{N_0}(n)]^T; \boldsymbol{t}(n) = [t_1(n), t_2(n), ..., t_{N_2}(n)]^T\right), \quad n = 1, 2, \ldots, N\right\}, \tag{1.4}$$

in which each $x_l(n)$ is the input to the l^{th} node in layer zero, and $t_k(n)$ is the *target* output for node k in the output layer (output *desired* in response to the corresponding input). The *computed* outputs of layer two (one) in response to $\boldsymbol{x}(n) = [x_1(n), ..., x_{N_0}(n)]^T$ are denoted $\boldsymbol{y}(n) = [y_1(n), ..., y_{N_2}(n)]^T$ ($\boldsymbol{y}'(n) = [y'_1(n), ..., y'_{N_2}(n)]^T$).

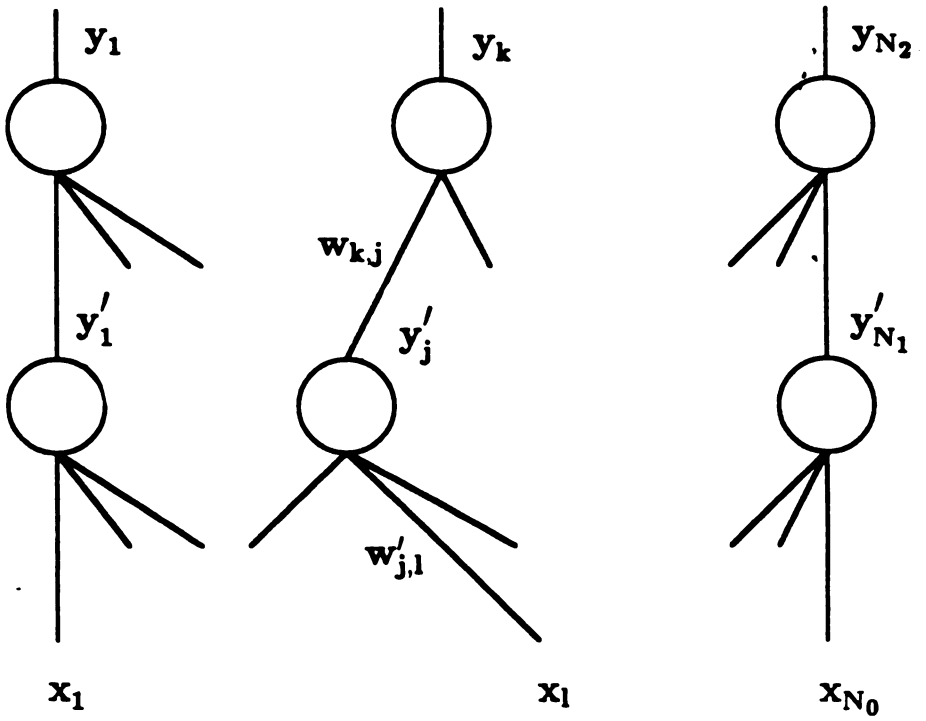

Figure 1.3: A two-layer network.

Vectors will be used when convenient, and scalars will be used when they simplify the presentation. Similarly to (1.1) we also define

$$u_k(n) \stackrel{\text{def}}{=} \sum_{j=1}^{N_1} w_{k,j} y'_j(n) = \boldsymbol{w}_k^T \boldsymbol{y}'(n). \tag{1.5}$$

Where $u_k(n)$ is the input to node k in the output layer in response to pattern n. $u'_j(n)$ is similarly defined as the input to node j in layer one.

1.3 Training Algorithms for FNNs

1.3.1 Performance Measures

There are many methods proposed for the training of FNNs. Training, in the most general sense, means to adjust the variable parameters of the network – the weights – so that the network can perform the desired task, be it pattern recognition or classification, a control problem, etc. The first problem encountered in training is in quantifying how well the network is performing the desired task. This quantification is done with a *performance measure.*

A performance measure is a function of the weights of the network. There are many in use. The most popular, and the one to be used in this research, is the sum of the squared errors between the desired outputs and the actual outputs. The training algorithm has the goal of adjusting the weights of the network so that the network performs well with respect to the performance measure used. Training algorithms can be divided into two overlapping groups. One group uses heuristics to search the weight space in order to optimize the performance measure, while the other group uses the performance measure in order to change the weights. Some training algorithms use both heuristics and the performance measure. One example of a training algorithm that uses a heuristic search over the parameter space is presented by Sun

and Grosky [8]. This algorithm uses random optimization and dynamic annealing to find a global minimum of the performance function. It does not compute the gradient of the performance function. As noted by Widrow [9], most algorithms that use the performance measure to change the weights can be further divided into two classes. The first class consists of algorithms which change the weights of the network to reduce the error between the desired response and the actual responce. Algorithms in this class are called error-correction rules. Algorithms from the second class are called gradient rules. These change the network weights along the gradient of the performance measure to reduce the mean-squared error. An example of an error correction rule is Rosenblatt's perceptron learning rule [10]. Training algorithms that use the gradient rule have become very common in recent research. There are many examples of these rules in the literature [11]. Because there are many similar algorithms, and many variations of the same algorithm, it would be very tedious to describe each. The following is a summary of the main gradient rule training algorithms, and those which are related to the present research.

The training problem for the two layer FNN is stated as follows: Given a set of N training patterns as in (1.4), find the weights which minimize the sum of squared errors, say E, between the target outputs and the actual outputs,

$$E = \sum_{n=1}^{N} \sum_{k=1}^{N_2} (t_k(n) - y_k(n))^2. \tag{1.6}$$

Minimizing E can also be viewed as the problem of finding a mapping from the input space, the space spanned by the $\boldsymbol{x}(n)$'s, to the output space, the space spanned by the $\boldsymbol{y}(n)$'s, so that E will be minimum.

It is the nonlinearity at the output of each node that gives the FNN the promise of solving previously intractable problems. However, it is also the nonlinearity that produces the problems in training. In order to appreciate some of the problems

encountered in training, consider the differences of the network with and without the nonlinearities. Removing the non-linearities from the outputs of the nodes will result in a linear network with the same structure as the FNN. If the network is linear, a single layer can achieve the same mapping as a multi-layer network, thus there is no advantage in using more than one layer, eg. [12, Ch.2]. In the linear case, finding the weights that minimize E is a well-known problem. Taking the partial derivative of E with respect to the weights and setting this equal to zero results in a set of linear equations which can be solved for the desired weights. For non-trivial cases in which there are at least as many independent training patterns as weights, the weights found in this manner are unique. This implies that there is only one minimum of the function E with respect to the weights.

Replacing the non-linearity at the output of each node changes many of the properties of the network. One difference is that one layer cannot map the inputs to all the outputs that a multilayer network can. As shown in [13], a one layer network, with the sigmoid replaced by the sgn function, can segment the input space into two regions by a hyper-plane. With multiple layers, more complex segmentation can be achieved. Multiple layers are thus useful, but pose more of a challenge to train than single layer networks. With one layer, once a set of training patterns is given, the inputs and outputs of every node are known. With multiple layers, the hidden layer "inputs" and "outputs" are not known and must be computed by the training algorithm.

Another difference between the linear network and the FNN is that there are generally multiple minima of the function E with respect to the weights. The occurence of multiple minima corresponds to the fact that the derivative of E is nonlinear in the weights. As before, we desire the weights that are the zeros of the equations resulting from taking the derivative of E with respect to the weights. Many methods for doing this appear in the literature. None of the methods, however, solves directly for the

weights. Some of the most popular methods for solving these non-linear equations are the gradient rule algorithms. All the gradient rule algorithms proceed in a similar manner. First, the partial derivative of E with respect to the weights at one training pattern, $\frac{\partial E}{\partial w}$, is evaluated at the present weights. Some method, depending on the algorithm, is then used to determine how to change the weights in order to reduce E. This is repeated for each training pattern. The weights can be changed with each training pattern, or after all the training patterns have been presented, depending on the algorithm. This constitutes one iteration through the training patterns. Many iterations are usually needed to find the desired weights. Thus, $\frac{\partial E}{\partial w}$ is evaluated at the new weights and the next training pattern, and the weights are again changed to reduce E. This is repeated until the partial derivative of E with respect to the weights is close to zero. The weights at which the algorithm stops are called the final weights.[2] Because there are multiple minima, the solution found may, or may not, be the global minimum. In practice many initial weights are tried and the weights giving the smallest E are chosen.

Because the non-linear nature of the problem requires an iterative approach, finding a solution will generally be much slower than in the linear case, where the weights can be found in one iteration or with one block computation. Finding a solution can be too slow in some cases, such as when the FNN is applied to speech processing [14, Ch.14]. New algorithms which produce a solution more rapidly than the techniques used currently would be of great value. This is the goal of the proposed research. Thus, some method to compare the "speed" of each algorithm is necessary.

[2]They are called final weights and not solution weights because these weights may not solve the proposed problem.

1.3.2 Training Efficiency

The speed at which an algorithm converges to a solution depends on two factors: the number of iterations and the complexity of each iteration. The complexity can be measured in terms of the number of operations, multiplications, additions etc., required. The number of iterations depends not only on the algorithm itself but on the initial weights of the system and the training data. The comparisons done in simulations to follow will eliminate these variables by using the same initial weights and the same training data for each algorithm.

Different algorithms do not always converge to the same set of final weights due to the different ways in which a true gradient system is approximated. However, even though the final weights may be different, it is useful to know the rate at which the algorithm converges to them. One way of doing this involves an *error function* [15]. This will be a function of the difference between the final weights, $\boldsymbol{w}^*$, and the weights at iteration n, $\boldsymbol{w}(n)$.

$$e(n) = \| w(n) - w^* \| \tag{1.7}$$

Where $\| \cdot \|$ is the euclidean norm. The rate that $e(n)$ goes to zero will indicate how fast the algorithm is converging to the final weights. If $e(n)$ can be written as a constant term multiplied by $e(n-1)$, $e(n) = Ke(n-1)$ then the algorithm is said to converge linearly. If $e(n) = Ke^2(n-1)$, then it is said to converge quadratically. In most cases a quadratically converging algorithm converges faster than a linearly converging one [15]. Of course not all algorithms converge linearly or quadratically, but it is useful to compare new algorithms with those known to converge linearly or quadratically.

1.3.3 The Back-Propagation Algorithm

The most prominent method for training FNNs is the back-propagation (BP) algorithm [17] [16]. Let $w(n)$ be any network weight at iteration n. Given a training pattern, it is possible to evaluate $\frac{\partial E}{\partial w(n)}$ for each weight in the network. A positive value for $\frac{\partial E}{\partial w(n)}$ indicates that reducing $w(n)$ will reduce E, conversely, a negative value of $\frac{\partial E}{\partial w(n)}$ indicates that an increase in $w(n)$ will reduce E. The back-propagation algorithm updates each weight at iteration n using the equation:

$$w(n) = w(n-1) - \varepsilon \frac{\partial E}{\partial w(n-1)}. \tag{1.8}$$

ε controls the rate of convergence, $0 < \varepsilon < 1$. $\frac{\partial E}{\partial w(n)}$ for weights in the $L-1$ layer depends on the weights in layer L. This is true for all layers. $\frac{\partial E}{\partial w(n)}$ for any weight in the network depends on the weights in the layers above.

The algorithm operates as follows: $\frac{\partial E}{\partial w(n)}$ is first evaluated and used to update each weight in the last layer. $\frac{\partial E}{\partial w(n)}$ is then evaluated for the weights in the $L-1$ layer using the updated weight estimates of layer L. In similar fashion, the weights in the remaining layers are updated. The process is then repeated with a new training pattern. The algorithm stops when the change to the weights is below a set threshold.

Assume the two layer network described above. The partial derivative with respect to a weight in the last layer, $w_{k,j}(n)$, for training pattern n is from (1.5) and (1.6),

$$\frac{\partial E}{\partial w_{k,j}(n)} = -2(t_k(n) - y_k(n)) \frac{\partial y_k(n)}{\partial u_k(n)} y'_j(n) \tag{1.9}$$

If $w'_{j,l}(n)$ is a weight associated with a node in the first layer, then for training pattern n

$$\frac{\partial E}{\partial w'_{j,l}(n)} = -2 \sum_{k=1}^{N_2} \left[(t_k(n) - y_k(n)) \frac{\partial y_k(n)}{\partial u_k(n)} w_{k,j} \right] \frac{\partial y'_j(n)}{\partial u'_j(n)} x_l(n) \tag{1.10}$$

As seen in (1.9), the partial derivative with respect to a weight in the last layer is a product of the error of node k, $t_k(n) - y_k(n)$, and the derivative of the output of the node with respect to the input and the node input. The partial derivative with respect to a weight in the first layer is similar, as seen in (1.10), with the error term replaced by the term $\sum_{k=1}^{N_2} \left[(t_k(n) - y_k(n))\frac{\partial y_k(n)}{\partial u_k(n)} w_{k,j}\right]$. This term can be interpreted as the "error " of node j in the first layer "back-propagated" through the upper layer.

It is clear from this analysis that one disadvantage of the BP algorithm is that by increasing the number of layers, the computational complexity is increased. For example, going from one layer to two more than doubles the complexity. For a network with M nodes in each layer, computing $\frac{\partial E}{\partial w}$ for a weight in the last layer takes three multiplications and two subtractions. Calculating $\frac{\partial E}{\partial w}$ for a weight in the next to last layer takes 3M+3 multiplications and 2M+1 subtractions. For a network with L layers and M nodes per layer, this algorithm requires $\mathcal{O}(3M^L)$ flops (a flop is one multiplication and one addition) per iteration.

Although BP as described by Rumelhart *et al.* [12], Parker [16], and Werbos [17] is not guaranteed to converge to a local minimum, changing the algorithm to use integration [18], does guarantee this convergence. This is a gradient system, and given any initial weights, will converge to a local minimum. Another disadvantage of BP is that it converges *linearly* to a minimum. Linear convergence, along with the high complexity prohibits real time training in many applications.

1.3.4 Techniques for Improving Convergence

Many new techniques have been presented to improve the speed of convergence [6] - [19] [20] [21]. One method has been to implement algorithms that use second order derivatives. Parker, who was among the first to derive the BP equations, later turned to second order algorithms to speed up convergence [22]. Many such techniques include some form of linearization [6], [19], [23]. If the weights are only changed

a small amount during each iteration, linearizing the equation for E with respect to the weights around the present weights is justified for each iteration. Thus the linearization must be performed every iteration.

One technique using linearization, the Azimi-Sadjadi *et al.* [6] algorithm, (A-S algorithm), uses linearized input, $\overline{y}'_j(n)$ and linearized output, $\overline{t}_k(n)$ target values. The equation to minimize is thus changed from (1.6) to

$$\overline{E} = \sum_{n=1}^{N} \sum_{k=1}^{N_2} \left(\overline{t}_k(n) - \left(\sum_{j=1}^{N_1} w_{k,j} \overline{y}'_j(n) \right) \right)^2 \tag{1.11}$$

In order to use equations of this form, the inputs and target outputs of every node must be known. Only the inputs to the first layer and targets for the last layer are given as training patterns, the other inputs and outputs must be calculated. The algorithm proceeds as follows: the input to the first layer, given by one training pattern n, is used to calculate the output of each layer at the present weights. The linearized input and output values are then calculated for the last layer. These values are used to update the second layer weights using the MIL-WRLS algorithm. The targets for the first layer, say $t'_j(n)$, are calculated using a "back propagated" error, similar to that used in the BP algorithm. These target outputs are used along with the training pattern inputs to calculate the linearized input and output target values for the first layer. As in the second layer, the weights of the first layer are updated using these linearized values and MIL-WRLS. The procedure is repeated for successive training pairs until the weight change is less than a preset threshold. In Chapter 2, we will interpret the A-S linearization process as one in which the linearized inputs and output for each node are chosen so that the error E of the nonlinear equations and the derivative of E with respect to the weights are the same for the linearized equations. Thus at the present weights

$$t_k(n) - y_k(n) = \bar{t}_k(n) - \sum_{j=1}^{N_1} w_{k,j}\bar{y}'_j(n) \quad \forall k \tag{1.12}$$

and

$$\frac{\partial}{\partial w_{k,j}} \sum_{k=1}^{N_2} (t_k(n) - y_k(n))^2 = \frac{\partial}{\partial w_{k,j}} \sum_{k=1}^{N_2} \left(\bar{t}_k(n) - \sum_{j=1}^{N_1} w_{k,j}\bar{y}'_j(n) \right)^2 \tag{1.13}$$

One disadvantage of the A-S algorithm is the complexity. In addition to using back-propagation to calculate the inner layer target values, it updates the weights using recursive equations. The algorithm requires $\mathcal{O}(3M^2)$ flops per node per iteration. This increases the complexity from $\mathcal{O}(M)$ for BP to $\mathcal{O}(3M^2)$. For a network with L layers and M nodes per layer, the algorithm requires an additional $\mathcal{O}(3M^3L)$ flops *per iteration* over BP. This algorithm however, converges faster than BP in simulations. Azimi-Sadjadi *et al.* attribute this mainly to the MIL-WRLS equations. Additionally, all the weights connected to a single node can be updated simultaneously instead of sequentially as in BP. In their research simulations using a two layer network, the number of iterations for this algorithm was an order of magnitude less than BP. Another disadvantage of this algorithm is that it uses recursive equations. The values to initialize these equations are not known practically. Also, previous equations used to update the weights decay exponentially, thus old linearized training patterns that are not useful for, and in fact may hinder, the updating of weights are still included in the equations.

Kollias and Anastassiou [23] also describe a recursive linearized algorithm. Their method is based on the Marquardt-Levenberg least squares optimization method. This method is designed to solve an approximation to the linear system

$$\boldsymbol{H}\Delta\boldsymbol{w} = -\boldsymbol{f} \tag{1.14}$$

where $\boldsymbol{H}$ is the hessian matrix containing the second derivatives of the error function

E with respect to the weights, and $\boldsymbol{f}$ is a vector consisting of the first derivatives of E with respect to the weights. This is basically Newton's method for determining the zeros of a non-linear equation in matrix form. Newton's method is known to converge quadratically [15] for initial weights close to a minimum. However, Newton's method can have convergence problems for poor initial weights. The Marquardt-Levenberg technique uses an approximation to $\boldsymbol{H}$. This allows the algorithm to converge for relatively poor initial weights and also retains the quadratic convergence when close to a minimum. This algorithm also uses back-propagation to determine the targets for the inner layers.

According to Kollias and Anastassiou the advantage of their algorithm is that it converges quadratically when close to a minimum. One disadvantage of the algorithm is that, like A-S, back-propagation is used for the inner layer training patterns. Also like A-S, it uses recursive equations which suffer from problems previously mentioned. Thus, this equation also has $\mathcal{O}(3M^2)$ flops per node per iteration. For a network with L layers and M nodes per layer, this algorithm requires $\mathcal{O}(3M^3L)$ per iteration in addition to $\mathcal{O}(3M^L)$ for BP. Steck, *et.al* [24] implement this technique and show that with parallel implementation, the additional computational requirements can be effectively reduced.

Another linearization algorithm, very similar to the Kollias and Anastassiou algorithm is presented by Ghiselli-Crippa and El-Jaroudi [25]. This algorithm also uses an approach based on an approximation of the Hessian matrix. The weight updating equations are:

$$\boldsymbol{w}(i+1) = \boldsymbol{w}(i) - \alpha \boldsymbol{H}^{-1}(i)\boldsymbol{F}(i) \tag{1.15}$$

where i is the iteration index, α is the step size, $\boldsymbol{w}$ is the vector of all the network weights, $\boldsymbol{F}$ is the gradient of the error E with respect to the network weights, and $\boldsymbol{H}$ is the approximation to the Hessian matrix. To simplify the Hessian matrix, the weights connected to each node are assumed to be independent. Simulations were

Table 1.1: Complexities per iteration and normalized complexities of FNN training algorithms. M is the number of nodes per layer. L is the number of layers.

	BP	A-S (QR-WRLS)	Layer-wise	Network-wise
complexity	$\mathcal{O}(3M^L)$	$\mathcal{O}(2.5M^3L + 3M^L)$	$\mathcal{O}(2.5M^{2L} + 3M^L)$	$\mathcal{O}(2.5M^{2L}L^2 + 3M^L)$
normalized complexity	∞	389.98	107.6	781.79

performed, training a FNN to classify speech as voiced, unvoiced, or silence. The error using this FNN for the classification was lower than that of a statistical decision classifier.

1.3.5 Complexity Measures

All of these linearization algorithms are more complex computationally than BP. Still, they offer faster convergence in the simulation examples presented by the respective authors. This implies that computational complexity is not a good method of judging the speed of an algorithm. To compare the speed of different algorithms we will use a "normalized" complexity, the complexity of the algorithm multiplied by a factor determined from the simulation studies in Chapter 4. The lower the normalized complexity, the faster the algorithm. Simulation studies were done comparing the algorithm developed in this dissertation with BP and the A-S algorithm. The simulations from the training of a 4-bit parity checker were used to determine the normalized complexities. The normalized complexity of the BP algorithm is shown to be ∞ since it failed to find the solution even once. Table 1.1 shows the relative complexities and the normalized complexities for these algorithms. The new layer-wise algorithm developed here has a clear advantage in speed over both BP and the A-S algorithm.

Chapter 2

New Training Algorithms

2.1 Introduction

This chapter is divided into two main sections, the first of which introduces a new class of linearized training algorithms for the FNN.

Because of the nonlinearities present in the FNN, all gradient descent learning algorithms perform some form of linearization around the present weights. The learning algorithms presented here are purely "linear" in the sense that the matrix vector equation to be solved has been transformed into the equation of a linear system. Accordingly, unlike other popular training algorithms that are not in this form, this linear algorithm and its potential variants will benefit from the well-understood theoretical properties of RLS and VLSI architectures for its implementation.

The second section describes implementation of these algorithms with QR-WRLS, a contemporary version of the conventional recursive least squares algorithm.

2.2 Linearized Training Algorithms

As described in the previous chapter, all training algorithms solve for the desired weights using several iterations through the training data. The gradient descent algorithms form a linear approximation to the error surface E at the present weights,

and change the weights in the direction which reduces the total error. The manner in which this approximation is made, and how it is implementated, account for the differences among the training algorithms. As can easily be seen from (1.6), the error is a function of all the weights. Typically the approximation to the error surface at a given iteration is done by taking the derivative of E with respect to a subset of all the weights and based on this gradient changing this subset to reduce E. This process is repeated for another subset until all the weights have been changed to reduce E. The most popular method of FNN training – BP – uses only one weight at a time for this approximation. In order to improve the rate of convergence and the final solution, newer techniques use more of the network weights, typically all the weights connected to one node, simultaneously to form an approximation to E at each iteration. Since E is a function of all the weights, it would seem better to simultaneously use as many of the weights as possible to reduce E. The class of algorithms presented here represents an improvement in the sense that the weights for an entire layer can be used to form an approximation to E and are updated simultaneously. Also in the special case that there is only one output, *all* the weights of the network can be updated simultaneously.

The theoretical development of the algorithm is first given for the case where all weights connected to a given node are updated simultaneously. This will be expanded to the case in which all the weights in the same *layer* are updated simultaneously. Finally the case in which all the weights in the network are updated simultaneously is presented. To simplify the presentation of each case, the development is done first for a two layer network, then for a general multilayer FNN.

2.2.1 Node-wise Weight Updating

2.2.1.1 Two Layer Network

First we wish to concentrate on the training of the weights in the final layer. Before

continuing, we note a simple fact which will reduce the number of details in our discussion. It is easy to show from (1.6) that if only the weights connected to output node k are allowed to change, with the other weights in the network held constant, then E is minimized by minimizing only the errors associated with this node, say E_k. Let us write E_k in a form which explicitly features the weights allowed to change,

$$E_k = \sum_{n=1}^{N} [t_k(n) - S(\boldsymbol{w}_k^T \boldsymbol{y}'(n))]^2. \tag{2.1}$$

Algorithms for finding the optimal solution, say $\boldsymbol{w}_k^*$, to this problem are well-known if the modeled output depends only upon a *linear* combination of the inputs using pattern-invariant (constant) weights. In the linear case $y_k(n) = S(\boldsymbol{w}_k \boldsymbol{y}'(n)) = \beta \boldsymbol{w}_k \boldsymbol{y}'(n)$, for some constant β (which can be taken as unity without loss of generality), and the error expression takes the form

$$E_k = \sum_{n=1}^{N} [t_k(n) - \boldsymbol{w}_k^T \boldsymbol{y}'(n)]^2. \tag{2.2}$$

The solution in the linear case is the solution to the classical linear least squares "normal equations" [7]. The solution of the normal equations can proceed in a variety of ways. It is also possible to arrive at the solution without explicitly forming the normal equations. This is the case, for example, when using the *least mean square* (LMS) (e.g. see [26] or [27]) algorithm, a recursive solution which amounts to "back-propagation" for a linear network. A second popular method is the conventional MIL-WRLS algorithm. A contemporary version of the latter will serve as a computational basis for the algorithm to be described in this dissertation, and MIL-WRLS is also the basis for the A-S algorithm to which we wish to relate the method of this dissertation. Appropriate description and formalism will be introduced as needed.

It is well-known that least squares estimation problems may be discussed in terms of their error surfaces, in this case the graph of E_k as a function of $\boldsymbol{w}_k$. Whatever

the form of the least square estimation algorithm, the ideal goal is to find the weight vector, say $\boldsymbol{w}_k^*$, corresponding to the global minimum of $E_k(\boldsymbol{w}_k)$. It is important to future developments to note that E_k depends not only on $\boldsymbol{w}_k$ but also upon the training patterns $\{(\boldsymbol{x}(n), t_k(n)),\ \ n \in [1, N]\}$ (see (2.1)). (In fact, since we have "frozen" the weights in the first layer, it is more appropriate in this case to view E_k as a function of $\boldsymbol{w}_k$ and the pairs $\{(\boldsymbol{y}'(n), t_k(n)), n \in [1, N]\}$.) Once the training patterns are fixed, the error function may be described as a surface over the N_1-dimensional hyperplane corresponding to the weights. Theoretically, the pairs $\{(\boldsymbol{y}'(n), t_k(n)), n \in [1, N]\}$ represent partial realizations of a two-dimensional stochastic process which generates them. In this sense

$$E_k(\boldsymbol{w}_k, \{(\boldsymbol{y}'(n), t_k(n)), n \in [1, N]\}) \tag{2.3}$$

is only a sample error surface. In a pure sense, we would like to find weights corresponding to the global minimum of $\mathcal{E}\{E_k(\boldsymbol{w}_k)\}$ where $\mathcal{E}$ denotes the expected value. We must be content, however, to work with the sample surface provided by the training data.

The point of the discussion above is to note that different algorithms construct and use different sample error surfaces from the data. With LMS (or back-propagation), error surfaces are sequentially constructed from *individual* training patterns, i.e., error surfaces of the form

$$E_k(\boldsymbol{w}_k, [\boldsymbol{y}'(n), t_k(n)]), \quad n = 1, 2, \ldots, N \tag{2.4}$$

are created, and for each n, the weights are moved in the direction of the negative gradient on that surface. The convergence properties are well-understood. MIL-WRLS[3], on the other hand, creates sequentially more refined error surfaces of the

[3]Of course, here we are speaking of a linear model identification.

form

$$E_k(\boldsymbol{w}_k, \{(\boldsymbol{y}'(j), t_k(j)), j \in [1,n]\}) \tag{2.5}$$

as n is incremented. At each step, if a weight update is computed, the solution corresponds to the unique minimum of the newly refined surface. We can appreciate, therefore, that even if we neglect nonlinearities, the estimation processes behave quite differently with respect to their error surface analysis.

The linearization technique adopted in this work can be explained in terms of the error surface analysis. The error surface over which we would like to find the (global) minimum by choice of weights is given by (2.1). Suppose we wish to construct a "linearized" error surface, say $\bar{E}_k$, which is "similar" in some sense to E_k in a neighborhood of the present weights. Recalling that E_k is a function not only of the weights, but also of the training patterns, the fundamental question is: *Can the pairs* $\{(\boldsymbol{y}'(n), \boldsymbol{t}(n)), n \in [1,N]\}$ *be modified in some sense, say* $(\boldsymbol{y}'(n), \boldsymbol{t}(n)) \rightarrow (\bar{\boldsymbol{y}}'_k(n), \bar{t}_k(n))$, *so that*

$$\begin{aligned} \bar{E}_k(\boldsymbol{w}_k, \{(\bar{\boldsymbol{y}}'_k(n), \bar{t}_k(n)), n \in [1,N]\}) &= \sum_{n=1}^{N} [\bar{t}_k(n) - \boldsymbol{w}_k^T \bar{\boldsymbol{y}}'_k(n)]^2 \\ \approx E_k(\boldsymbol{w}_k, \{(\boldsymbol{y}'(n), t_k(n)), n \in [1,N]\}) &= \sum_{n=1}^{N} [t_k(n) - S(\boldsymbol{w}_k^T \boldsymbol{y}'(n))]^2 \end{aligned} \tag{2.6}$$

in some neighborhood of the present weights? The answer to this question is the key theoretical development described in the following paragraphs.

In the ensuing discussion, the notation $\boldsymbol{w}_k^*$ will be used to designate a *local* minimum of E_k. Ideally, $\boldsymbol{w}_k^*$ will be the *global* minimum, but we have no way of assuring this. The objective is to find, by means of a "linear" algorithm, a close approximation to $\boldsymbol{w}_k^*$.

The algorithm to be described proceeds in iterations, indexed by $i = 1, 2, \ldots$. Each iteration represents one complete training cycle through the N training patterns.

Suppose that a weight vector estimate $\boldsymbol{w}_k(i-1)$ results from iteration $i-1$. In iteration i, by manipulation of the data, we work with a "linearized" error surface, $\bar{E}_k$, which is similar to the nonlinear surface in the neighborhood of $\boldsymbol{w}_k(i-1)$. The similarity follows from two criteria:

1. $\bar{E}_k(\boldsymbol{w}_k(i-1), \{(\bar{\boldsymbol{y}}'_k(n), \bar{t}_k(n)), n \in [1,N]\}) = E_k(\boldsymbol{w}_k(i-1), \{(\boldsymbol{y}'_k(n), t_k(n)), n \in [1,N]\})$;
2. $\frac{\partial E_k}{\partial \boldsymbol{w}_k}\Big]_{\boldsymbol{w}_k=\boldsymbol{w}_k(i-1)} = \frac{\partial E_k}{\partial \boldsymbol{w}_k}\Big]_{\boldsymbol{w}_k=\boldsymbol{w}_k(i-1)}$.

The first task is to manipulate the pairs $\{(\bar{\boldsymbol{y}}'_k(n), \bar{t}_k(n)), n \in [1,N]\}$ so that these criteria hold. This is accomplished as follows. It follows from Criterion 1 that

$$\sum_{n=1}^{N}(t_k(n) - y_k(n))^2 = \sum_{n=1}^{N}(\bar{t}_k(n) - \boldsymbol{w}_k^T(i-1)\bar{\boldsymbol{y}}'_k(n))^2. \tag{2.7}$$

By letting

$$t_k(n) - y_k(n) = \bar{t}_k(n) - \boldsymbol{w}_k^T(i-1)\bar{\boldsymbol{y}}'_k(n), \tag{2.8}$$

or

$$\bar{t}_k(n) = (t_k(n) - y_k(n)) + \boldsymbol{w}_k^T(i-1)\bar{\boldsymbol{y}}'_k(n), \tag{2.9}$$

for each n, Criterion 1 is met. Now we take the partial derivatives required in Criterion 2. For the "nonlinear" error,

$$\begin{aligned}\frac{\partial E_k}{\partial \boldsymbol{w}_k}\Big]_{\boldsymbol{w}_k=\boldsymbol{w}_k(i-1)} &= -2\sum_{n=1}^{N}(t_k(n) - y_k(n))\dot{S}(u_k(n))\boldsymbol{y}'(n) \\ &= -2\sum_{n=1}^{N}(t_k(n) - y_k(n))\dot{S}(\boldsymbol{w}_k^T(i-1)\boldsymbol{y}'(n))\boldsymbol{y}'(n) \end{aligned} \tag{2.10}$$

where

$$\dot{S}(u_k(n)) \stackrel{\text{def}}{=} \frac{dS(\alpha)}{d\alpha}\Big]_{\alpha=u_k(n)}. \tag{2.11}$$

All inputs and outputs in this and similar expressions are those associated with weights $\boldsymbol{w}_k(i-1)$ (or the "current" set of weights around which linearization is tak-

ing place), but we will avoid writing $u_k(i-1,n)$, for example, for simplicity. For the "linear" error,

$$\left.\frac{\partial \bar{E}}{\partial \boldsymbol{w}_k}\right]_{\boldsymbol{w}_k=\boldsymbol{w}_k(i-1)} = -2\sum_{n=1}^{N}(\bar{t}_k(n) - \boldsymbol{w}_k^T(i-1)\bar{\boldsymbol{y}}_k'(n))\bar{\boldsymbol{y}}_k'(n). \tag{2.12}$$

Equating (2.10) and (2.12), in light of (2.8) we have

$$\bar{\boldsymbol{y}}_k'(n) = \dot{S}(\boldsymbol{w}_k^T(i-1)\boldsymbol{y}'(n))\boldsymbol{y}'(n). \tag{2.13}$$

All quantities needed to compute the modified pair $(\bar{t}_k(n), \bar{\boldsymbol{y}}_k'(n))$ are known or can be calculated at pattern n in iteration i. This procedure is repeated for each k (output node).

Before extending the analysis down to layer one, let us ponder the significance of what we have done. By modifying the data pairs, we have created a "linear" error surface which is similar to the "nonlinear" one in the neighborhood of $\boldsymbol{w}_k(i-1)$. In particular, the error surfaces match at that point, and their gradients are identical with respect to the weight vectors. We can find the $\boldsymbol{w}_k$ which minimizes $\bar{E}_k$ by simple *linear* least squares processing of the modified data $\{(\bar{t}_k(n), \bar{\boldsymbol{y}}'(n)),\ n \in [1,N]\}$. The linear estimate will correspond to a minimum of the error surface $\bar{E}_k$ which need not be near a minimum of E_k. However, because the error surfaces and the gradients match with respect to the weight vector of node k, if the weight change is small enough, the weight change will be in the direction of decreasing E_k. Accordingly, the linear weights must be constrained to remain in a reasonably small neighborhood of $\boldsymbol{w}_k(i-1)$. Because E_k is reduced at each iteration, it is to be expected that a minimum of E will be reached by repeating this procedure. In turn, this implies convergence to the "nonlinear" solution for the weights, using purely linear techniques.

Let us now move down to the lower layer and consider the estimation of the weights $\left\{\boldsymbol{w}_j',\ \ j \in [1,N_1]\right\}$. Again let us focus on a single node, say node l. However,

we must now optimize $\boldsymbol{w}_l'$ with respect to the *entire* external error, E, since all nodes in the upper layer are affected by these weights. Suppose that we are working on the i^{th} cycle through the training patterns and that all weights in the upper layer are fixed at their newly updated values $\{\boldsymbol{w}_k(i),\ k \in [1, N_2]\}$. Taking the derivative of E with respect to $\boldsymbol{w}_j'$,

$$\frac{\partial E}{\partial \boldsymbol{w}_j'} = -2 \sum_{n=1}^{N} \sum_{k=1}^{N_2} (t_k(n) - y_k(n)) \dot{S}(u_k(n)) w_{k,j}(i) \dot{S}([\boldsymbol{w}_j']^T \boldsymbol{x}(n)) \boldsymbol{x}(n) \tag{2.14}$$

where $w_{k,j}(i)$ denotes j^{th} element in weight vector $\boldsymbol{w}_k(i)$ (weight on connection from node j in layer one to node k in layer two). This expression can be written

$$\frac{\partial E}{\partial \boldsymbol{w}_j'} = -2 \sum_{n=1}^{N} (t_j'(n) - y_j'(n)) \dot{S}([\boldsymbol{w}_j']^T \boldsymbol{x}(n)) \boldsymbol{x}(n) \tag{2.15}$$

where $t_j'(n)$ is called the *target value* for inner node j and is defined such that

$$(t_j'(n) - y_j'(n)) = \sum_{k=1}^{N_2} (t_k(n) - y_k(n)) \dot{S}(u_k(n)) w_{k,j}(i). \tag{2.16}$$

The quantity on the right side of (2.16) is commonly called the *back-propagated error* for node j. The solution sought, say $\boldsymbol{w}_j'^*$, is one for which

$$\left. \frac{\partial E}{\partial \boldsymbol{w}_j'} \right]_{\boldsymbol{w}_j' = \boldsymbol{w}_j'^*} = 0. \tag{2.17}$$

In the top layer, for node k we sought $\boldsymbol{w}_k^*$ such that

$$\left. \frac{\partial E}{\partial \boldsymbol{w}_k} \right]_{\boldsymbol{w}_k = \boldsymbol{w}_k^*} = 0. \tag{2.18}$$

With reference to (2.10), it is clear that the present optimization problem is equivalent to the ones encountered at the upper nodes. In particular, the same linearization

considerations can be applied to obtain modified input and target values, say

$$(t'_j(n), \boldsymbol{x}(n)) \rightarrow (\bar{t}'_j(n), \bar{\boldsymbol{x}}_j(n)) \tag{2.19}$$

and the set of layer one weights $\boldsymbol{w}'_j(i)$ computed accordingly for each j.

2.2.1.2 L Layer Network

This algorithm can be extended in a straightforward, though notationally cumbersome, manner to any number of layers. Here we will use parenthetical superscripts instead of primes to indicate the layer. In order to simplify notation, the dependence on the iteration number will be dropped. Thus, $y_j^{(l)}$ will be used to denote the output of node j in layer l. Similarly, $w_{j,p}^{(l)}$ will be used to denote the weight connecting node p in layer $l-1$ to node j in layer l. Assuming a network with L layers, let us update the weights connected to node i in layer l, $\boldsymbol{w}_i^{(l)}$, $l \in [1, 2, \ldots, L]$. We have already presented the cases of $l = L$ and $l = L-1$. Taking the derivative of E with respect to $\boldsymbol{w}_i^{(l)}$,

$$\begin{aligned}
\frac{\partial E}{\partial \boldsymbol{w}_i^{(l)}} = -2 \sum_{n=1}^{N} \sum_{k=1}^{N_L} \Bigg(& (t_k(n) - y_k^{(L)}(n)) \dot{S}(u_k^{(L)}(n)) \Bigg(\sum_{j=1}^{N_{L-1}} (w_{k,j}^{(L)} \dot{S}(u_j^{(L-1)}(n)) \\
& \Bigg(\sum_{p=1}^{N_{L-2}} (w_{j,p}^{(L-1)} \dot{S}(u_p^{(L-2)}(n)) \cdots \\
& \sum_{m=1}^{N_{l+1}} w_{p,m}^{(l+2)} \dot{S}(u_m^{(l+1)}(n)) w_{m,i}^{(l+1)} \Bigg) \Bigg) \Bigg) \Bigg) \dot{S}(u_i^{(l)}(n)) \boldsymbol{y}^{(l-1)}(n)
\end{aligned} \tag{2.20}$$

Similarly to (2.15), this expression can be written

$$\frac{\partial E}{\partial \boldsymbol{w}_i^{(l)}} = -2 \sum_{n=1}^{N} (t_i^{(l)}(n) - y_i^{(l)}(n)) \dot{S}([\boldsymbol{w}_i^{(l)}]^T \boldsymbol{y}^{(l-1)}(n)) \boldsymbol{y}^{(l-1)}(n) \tag{2.21}$$

where $t_i^{(l)}$ is the target value for node i layer l, and is defined by

$$
\begin{aligned}
(t_i^{(l)}(n) - y_i^{(l)}(n)) \;=\; & \sum_{k=1}^{N_L} \Big((t_k(n) - y_k^{(L)}(n)) \dot{S}(u_k^{(L)}(n)) \\
& \left(\sum_{j=1}^{N_{L-1}} (w_{k,j}^{(L)} \dot{S}(u_j^{(L-1)}(n)) \left(\sum_{p=1}^{N_{L-2}} (w_{j,p}^{(L-1)} \dot{S}(u_p^{(L-2)}(n)) \cdots \right. \right. \\
& \left. \left. \sum_{m=1}^{N_{l+1}} w_{p,m}^{(K+2)} \dot{S}(u_m^{(K+1)}(n)) w_{m,i}^{(K+1)} \right) \right) \Big)
\end{aligned}
\tag{2.22}
$$

Again the optimization problem is equivalent to the ones encountered at the upper nodes.

Before continuing, let us note the relationship to the A-S algorithm noted above. In fact, to this point in the discussion, the methods are nearly equivalent, though derived from different starting points. The A-S algorithm proceeds by replacing the nonlinearity $S(\cdot)$ for the node to be updated by a linear approximation, say $\hat{S}(\cdot)$, consisting of the first two terms of a Taylor series around the "present" value of the node's input. For example, suppose the k^{th} output node is to be linearized with respect to the n^{th} training pattern. Let $\hat{w}_{k,j}$ denote the present value of weight $w_{k,j}$. Then,

$$
\begin{aligned}
S(u) \;\approx\; & \hat{S}(u) = \dot{S}\left(\sum_{j=1}^{N_1} \hat{w}_{k,j} y_j'(n) \right) \left(u - \sum_{j=1}^{N_1} \hat{w}_{k,j} y_j'(n) \right) + S\left(\sum_{j=1}^{N_1} \hat{w}_{k,j} y_j'(n) \right) \\
=\; & \dot{S}\left(\sum_{j=1}^{N_1} \hat{w}_{k,j} y_j'(n) \right) u + \left[S\left(\sum_{j=1}^{N_1} \hat{w}_{k,j} y_j'(n) \right) - \dot{S}\left(\sum_{j=1}^{N_1} \hat{w}_{k,j} y_j'(n) \right) \sum_{j=1}^{N_1} \hat{w}_{k,j} y_j'(n) \right] \\
\stackrel{\text{def}}{=}\; & K_k(n) u + b_k(n).
\end{aligned}
\tag{2.23}
$$

Azimi-Sadjadi *et al.* [6] recognized that by using this approximation in (2.10), the optimization problem became equivalent to a set of linear least square error normal equations *if* the data were modified according to (2.9) and (2.13). Therefore, by quite different means, the theoretical developments arrive at the same set of linear

equations to be solved.

2.2.2 Layer-wise Weight Updating

2.2.2.1 Two Layer Network

The method used above – finding new linearized training patterns to update the network weights – can be extended to update all the weights of one layer simultaneously. In the previous section, E and $\bar{E}$ were used directly to calculate the linearized training patterns. This is useful from a theoretical point of view, it shows exactly why the linearized training patterns can be used to calculate the gradient of the system, but is cumbersome notationally. As noted in the previous section, the linearized error surface $\bar{E}$ can be constructed by replacing the nonlinearity $S(\cdot)$ by a linear approximation. Because using this approximation simplifies notation and is equivalent to using E and $\bar{E}$ directly, in this section we will use this latter approach for calculating the linearized training patterns.

Consider again the general L layer network. Suppose that the weights connected to one or more nodes in layer l are to be updated simultaneously[4]. This may include as few as one, and as many as all, nodes in layer l. Denote the set of such selected nodes in layer l by $\mathcal{N}$. Denote by $\mathcal{M}$ the set of all nodes above layer l to which any node in $\mathcal{N}$ is connected, directly or indirectly. Let all weights *not* connected to nodes in $\mathcal{N}$ and $\mathcal{M}$ be fixed at present values[5]. As shown above, the linearized error surface is constructed by replacing the nonlinearity $S(\cdot)$ for each node in $\mathcal{N}$ and $\mathcal{M}$ by the linear approximation, $\hat{S}(\cdot)$. In fact, any node not in $\mathcal{N}$ or $\mathcal{M}$ may also be linearized with no effect on the solution. Therefore, we may assume without loss of generality that the entire network is "linearized," even if only a portion of the weights is to be

[4]If any weight connected to a node is to be updated, then every weight connected to that node must be updated. This "constraint" is ordinarily beneficial, since it implies the ability to simultaneously update more than one weight.

[5]In certain cases it is possible to update weights in different layers simultaneously. We discuss one case at the end of this section.

updated.

Let us now return to the two layer network. Suppose we wish to update all weights in the output layer simultaneously. We must linearize all output nodes (and may arbitrarily linearize any other nodes). For node k in the output layer, the output in response to input n is computed as in (1.2), which we repeat here for convenience.

$$y_k(n) = S(u) = S\left(\sum_{j=1}^{N_1} w_{k,j} y'_j\right) \tag{2.24}$$

As before, we let $\bar{y}_k(n)$ represent the output of node k after $S(u_k(n))$ has been replaced by $\hat{S}(u_k(n)) = K_k u_k(n) + b_k$. Accordingly,

$$\bar{y}_k(n) = K_k(n)\left[\sum_{j=1}^{N_1} w_{k,j} y'_j(n)\right] + b_k(n) \quad \text{or} \quad \bar{z}_k(n) = K_k(n)\left[\sum_{j=1}^{N_1} w_{k,j} y'_j(n)\right] \tag{2.25}$$

with

$$\bar{z}_k(n) \stackrel{\text{def}}{=} \bar{y}_k(n) - b_k(n). \tag{2.26}$$

We speak of the rightmost form in (2.25) as descriptive of a *linearized node* because the output is a purely linear combination of the inputs to the node. The network with all appropriate nodes linearized will be called the *linearized network.* Since $\bar{y}_k(n) = y_k(n)$ at the present weights, the error at the k^{th} node will be the same for the linearized and original network if the target value for $\bar{z}_k(n)$, say $\bar{t}_k(n)$, is taken to be

$$\bar{t}_k(n) \stackrel{\text{def}}{=} t_k(n) - b_k(n) \tag{2.27}$$

and the "linearized" inputs to node k at pattern n are

$$\bar{x}_{k,j}(n) \stackrel{\text{def}}{=} K_k(n) y'_j(n), \quad j = 1, 2, \ldots, N_1. \tag{2.28}$$

Note that the linearized inputs are dependent upon k, so that we have effectively

increased the number of training pairs by a factor of N_2.

The problem has effectively been reduced to one of estimating weights for a single-layer linear network. In order to simultaneously update all the weights in the output layer, the system of $N \times N_2$ equations

$$\bar{t}_k(n) = \sum_{j=1}^{N_1} \bar{x}_{k,j}(n) w_{k,j}, \quad k = 1, 2, \ldots, N_2 \quad n = 1, 2, \ldots, N \tag{2.29}$$

must be solved for the least square estimate of the $N_1 \times N_2$ weights $w_{k,j}$, $k \in [1, N_2]$ $j \in [1, N_1]$. However, since all weights in the hidden layer are fixed, the outputs $y'_j(n)$ are independent of k. This means that the equations indexed by different values of k are independent of one another, and the sets of weights connected to different outputs may be updated independently. In the output layer, therefore, there is no theoretical difference between layer-wise and node-wise updating. To prove this we need to show that the $\bar{t}_k(n)$ of (2.9) and $\bar{\boldsymbol{y}}'_k(n)$ of (2.13) are the same as the $\bar{t}_k(n)$ and $\bar{x}_{k,j}(n)$ of (2.29). Substituting for $K_k(n)$ in (2.28) gives

$$\begin{aligned} \bar{x}_{k,j}(n) &= \dot{S}(u_k(n)) y'_j(n), \quad j = 1, 2, \ldots, N_1 \\ &= \dot{S}(\boldsymbol{w}_k^T \boldsymbol{y}'(n)) y'_j(n). \end{aligned} \tag{2.30}$$

In vector form this equation is

$$\bar{\boldsymbol{x}}_{k,j}(n) = \dot{S}(\boldsymbol{w}_k^T \boldsymbol{y}'(n)) \boldsymbol{y}'(n) = \bar{\boldsymbol{y}}'_k(n). \tag{2.31}$$

Substituting for b_k in (2.27) we get,

$$\bar{t}_k(n) = t_k(n) - \left(y_k(n) - K_k(n) \sum_{j=1}^{N_1} w_{k,j} y'_j(n) \right) \tag{2.32}$$

$$
\begin{aligned}
&= (t_k(n) - y_k) - \sum_{j=1}^{N_1} w_{k,j} K_k(n) y'_j(n) \\
&= (t_k(n) - y_k) - \boldsymbol{w}_{k,j}^T K_k(n) \boldsymbol{y}'(n) \\
&= (t_k(n) - y_k) - \boldsymbol{w}_{k,j}^T \bar{\boldsymbol{y}}'_k(n)
\end{aligned}
$$

Which is equivalent to (2.9). Thus the linearized inputs and outputs of the layer-wise and node-wise weight updating are the same for the last layer. This is not true at lower layers, however, as we now show for the hidden layer of the present network.

In order to update all weights in the hidden layer simultaneously, the weights in the output layer are fixed and *all* nodes in the network must be linearized. The outputs of the hidden layer with $S(\cdot)$ replaced by $\hat{S}(\cdot)$ are given by

$$
\bar{y}'_j(n) = K'_j(n)[\sum_{p=1}^{N_0} w'_{j,p} x_p(n)] + b'_j(n) \quad j = 1, 2, \ldots, N_1. \tag{2.33}
$$

Substituting (2.33) in the leftmost expression in (2.25) results in

$$
\bar{y}_k(n) - [\sum_{j=1}^{N_1} K_k(n) w_{k,j} b'_j(n) + b_k(n)] = \sum_{j=1}^{N_1} \sum_{p=1}^{N_0} [K_k(n) w_{k,j} K'_j(n) x_p(n)] w'_{j,p}. \tag{2.34}
$$

As above, we can now view the problem as one of training a single-layer linear mapping with target outputs

$$
\vec{t}'_k(n) = t_k(n) - [\sum_{j=1}^{N_1} K_k(n) w_{k,j} b'_j(n) + b_k(n)] \tag{2.35}
$$

and inputs

$$
\bar{x}'_{k,j,p}(n) = K_k(n) w_{k,j} K'_j(n) x_p(n). \tag{2.36}
$$

The weight estimates for $w'_{j,p}$; $j \in [1, N_1]$, $p \in [1, N_0]$ comprise the least square error

solution to the system of equations

$$\vec{t}_k'(n) = \sum_{j=1}^{N_1} \sum_{p=1}^{N_0} \bar{x}_{k,j,p}'(n) w_{j,p}' \quad k = 1, 2, \ldots, N_2 \quad n = 1, 2, \ldots, N. \tag{2.37}$$

Unlike the output layer, we see that the problem cannot be decomposed into separate solutions for sets of weights connected to individual nodes in the hidden layer. This is a reflection of the fact that all weights in the hidden layer are coupled through their "mixing" in the output layer. This means that the simultaneous solution for all weights in the hidden layer should be beneficial with respect to a node-wise solution. Indeed we will find this to be the case in the experiments.

Note that, for a *fixed* k, the inputs to the linearized network, $\bar{x}'(n)$, $n \in [1, N]$, are most conveniently viewed as two-dimensional (indexed by couples (j, p)). There are N such "grid" inputs for each k, paired with the N values of $\vec{t}_k'(n)$. If there were two hidden layers in the network, we would find that there would be *three* effective inputs. Although the number of subscripts continues to increase with increasing layers, only the subscript of the output node, and the subscripts of the weight with which the linearized input is associated are used. (see (2.44)). The other subscripts are "summed" out of the equation. Further, it is noted that the role of k in (2.37) is somewhat superfluous. In principle, the index is used to keep track of which of N_2 outputs in the linearized network is being considered. However, the training pairs $(\vec{t}_k'(n); \bar{x}_{k,1,1}'(n), \ldots, \bar{x}_{k,N_1,N_0}'(n))$, $k \in [1, N_2]$, $n \in [1, N]$, can be reindexed by mapping pairs $(k, n) \rightarrow i$ so that the training pairs may be written $(\vec{t}'(i); \bar{x}_{1,1}'(i), \ldots, \bar{x}_{N_1,N_0}'(i))$, $i \in [1, N \times N_2]$. Of course, an identical system of equations to (2.37) results, but the linearized network may be viewed as a *single output* linear layer with $N \times N_2$ training pairs.

Updating of some *subset* of the weights in the hidden layer (in particular, "node-wise" as in the A-S algorithm) is tantamount to solving the subsystem of (2.37)

corresponding to the desired weights, introducing the updated values into the system, solving for the next desired subset, etc. Clearly this will result in a different solution than the simultaneous solution. In terms of the error surfaces, this process consists of continually updating the error surface as "partial" information becomes available, then moving in the direction of the gradient with respect to a new subset of weights in the updated surfaces. Intuitively, movement "at once" with respect to the "complete" gradient would seem to be a preferable procedure. Indeed, the later operation corresponds to the simultaneous updating.

The linearization allows us to approximate the error surface of the nonlinear system for only a small neighborhood around the present weights. Because of the criteria used to construct $\bar{E}$, the weights will be changed in the *direction* of the true gradient in the nonlinear space, but will move to the minimum of $\bar{E}$ which may be quite far from the neighborhood over which $E \approx \bar{E}$. As in the node-wise weight updating, the weights must be allowed to change only a small amount using the training patterns of the linearized system. If the linearized procedure results in a large change of weights, measures must be taken to decrease the alteration. The updating procedure is repeated until changing the weights does not result in a decrease in error. The algorithm proceeds as follows: linearize the system around the present weights, change the weights by a small amount to decrease error, then repeat the procedure. This is done until changing the weights does not decrease the error or a maximum on the number of iterations is reached.

2.2.2.2 L Layer Network

The general case of an L layer network is a straightforward extension of the two layer case. As in the previous section, let a parenthetical superscript indicate the layer. After linearizing all the nodes, the output equation for node k in layer L in terms of the outputs of layer $L-1$ is,

$$\bar{y}_k^{(L)}(n) = K_k^{(L)}(n)\left(\sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)} y_{j_1}^{(L-1)}(n)\right) + b_k^{(L)}(n) \tag{2.38}$$

This is equivalent to (2.25) and can be used to solve for layer L weights. Extending this down one more layer, and writing $\bar{y}_k^{(L)}(n)$ in terms of the outputs of layer $L-2$,

$$\begin{aligned}
\bar{y}_k^{(L)}(n) &= K_k^{(L)}(n)\left(\sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)}\left(K_{j_1}^{(L-1)}(n)\right.\right. \\
&\qquad \left.\left.\sum_{j_2=1}^{N_{L-2}} w_{j_1,j_2}^{(L-1)} y_{j_2}^{(L-2)}(n) + b_{j_1}^{(L-1)}(n)\right)\right) + b_k^{(L)}(n) \\
&= \sum_{j_1=1}^{N_{L-1}}\sum_{j_2=1}^{N_{L-2}} K_k^{(L)}(n) w_{k,j_1}^{(L)} K_{j_1}^{(L-1)}(n) w_{j_1,j_2}^{(L-1)} y_{j_2}^{(L-2)}(n) \\
&+ b_k^{(L)}(n) + K_k^{(L)} \sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)} b_{j_1}^{(L-1)}(n)
\end{aligned} \tag{2.39}$$

Continuing the process for another layer gives,

$$\begin{aligned}
\bar{y}_k^{(L)}(n) &= \sum_{j_1=1}^{N_{L-1}}\sum_{j_2=1}^{N_{L-2}} K_k^{(L)}(n) w_{k,j_1}^{(L)} K_{j_1}^{(L-1)}(n) w_{j_1,j_2}^{(L-1)} \\
&\qquad \left(K_{j_2}^{(L-2)}(n)\sum_{j_3=1}^{N_{L-3}} w_{j_2,j_3}^{(L-2)} y_{j_3}^{(L-3)}(n) + b_{j_2}^{(L-2)}(n)\right) \\
&+ b_k^{(L)}(n) + K_k^{(L)} \sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)} b_{j_1}^{(L-1)}(n) \\
&= \sum_{j_1=1}^{N_{L-1}}\sum_{j_2=1}^{N_{L-2}}\sum_{j_3=1}^{N_{L-3}} \left(K_k^{(L)}(n) w_{k,j_1}^{(L)} K_{j_1}^{(L-1)}(n) w_{j_1,j_2}^{(L-1)} K_{j_2}^{(L-2)}(n) w_{j_2,j_3}^{(L-2)} y_{j_3}^{(L-3)}(n)\right) \\
&+ b_k^{(L)}(n) + K_k^{(L)} \sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)}\left(b_{j_1}^{(L-1)}(n) + K_{j_1}^{(L-1)} \sum_{j_2=1}^{N_{L-2}} w_{j_1,j_2}^{(L-1)}\left(b_{j_2}^{(L-2)}(n)\right)\right)
\end{aligned} \tag{2.40}$$

It is easy to extend this to any number of layers. Let $l \in [1, L]$. Then the equations to update the weights in this layer are,

$$\begin{aligned}
\bar{y}_k^{(L)}(n) &= \sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \cdots \sum_{j_{L-l-1}=1}^{N_{l+1}} \sum_{j_{L-l}=1}^{N_l} \sum_{j_{L-l+1}=1}^{N_{l-1}} \Big(K_k^{(L)}(n) w_{k,j_1}^{(L)} K_{j_1}^{(L-1)}(n) \\
&\quad w_{j_1,j_2}^{(L-1)} \cdots K_{j_{L-l}}^{(l)}(n) w_{j_{L-l},j_{L-l+1}}^{(l)} y_{j_{L-l+1}}^{(l-1)}(n) \Big) \\
&+ b_k^{(L)}(n) + K_k^{(L)} \sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)} \Big(b_{j_1}^{(L-1)}(n) \\
&+ K_{j_1}^{(L-1)} \sum_{j_2=1}^{N_{L-2}} w_{j_1,j_2}^{(L-1)} \left(b_{j_2}^{(L-2)}(n) + \cdots K_{j_{L-l-1}}^{(l+1)} \sum_{j_{L-l}=1}^{N_l} w_{j_{L-l-1},j_{L-l}}^{(l+1)} b_{j_{L-l}}^{(l)} \right) \Big)
\end{aligned} \tag{2.41}$$

$$\begin{aligned}
\bar{y}_k^{(L)}(n) &- \Bigg[b_k^{(L)}(n) + K_k^{(L)} \sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)} \Big(b_{j_1}^{(L-1)}(n) \\
&+ K_{j_1}^{(L-1)} \sum_{j_2=1}^{N_{L-2}} w_{j_1,j_2}^{(L-1)} \left(b_{j_2}^{(L-2)}(n) + \cdots K_{j_{L-l-1}}^{(l+1)} \sum_{j_{L-l}=1}^{N_l} w_{j_{L-l-1},j_{L-l}}^{(l+1)} b_{j_{L-l}}^{(l)} \right) \Big) \Bigg] \\
&= \sum_{j_{L-l}=1}^{N_l} \sum_{j_{L-l+1}=1}^{N_{l-1}} \Bigg(\Bigg[\sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \cdots \sum_{j_{L-l-1}=1}^{N_{l+1}} K_k^{(L)}(n) w_{k,j_1}^{(L)} K_{j_1}^{(L-1)}(n) \\
&\quad w_{j_1,j_2}^{(L-1)} \cdots K_{j_{L-l-1}}^{(l+1)}(n) y_{j_{L-l+1}}^{(l)}(n) \Bigg] w_{j_{L-l},j_{L-l+1}}^{(l)} \Bigg)
\end{aligned} \tag{2.42}$$

Similarly to above, this can be viewed as a single layer linear mapping with target outputs

$$\begin{aligned}
\bar{t}_k^{(l)}(n) &= t_k^{(L)}(n) - \Bigg[b_k^{(L)}(n) + K_k^{(L)} \sum_{j_1=1}^{N_{L-1}} w_{k,j_1}^{(L)} \Bigg(b_{j_1}^{(L-1)}(n) + K_{j_1}^{(L-1)} \sum_{j_2=1}^{N_{L-2}} w_{j_1,j_2}^{(L-1)} \\
&\quad \left(b_{j_2}^{(L-2)}(n) + \cdots K_{j_{L-l-1}}^{(l+1)} \sum_{j_{L-l}=1}^{N_l} w_{j_{L-l-1},j_{L-l}}^{(l+1)} b_{j_{L-l}}^{(l)} \right) \Bigg) \Bigg]
\end{aligned} \tag{2.43}$$

and inputs

$$\begin{aligned}
\bar{x}_{k,j_{L-l},j_{L-l+1}}^{(l)}(n) &= \Bigg[\sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \cdots \sum_{j_{L-l-1}=1}^{N_{l+1}} K_k^{(L)}(n) w_{k,j_1}^{(L)} K_{j_1}^{(L-1)}(n) \\
&\quad w_{j_1,j_2}^{(L-1)} \cdots K_{j_{L-l-1}}^{(l+1)}(n) y_{j_{L-l+1}}^{(l)}(n) \Bigg]
\end{aligned} \tag{2.44}$$

In the same manner as described above these linearized inputs and outputs can be used to update the weights in layer l.

2.2.3 Network-wise Weight Updating

2.2.3.1 Two Layer Network

For the same reason that simultaneous *layer*-wise estimation of weights is beneficial, we should expect even more benefit from complete *network* updating if such were possible. It follows from the developments above that entire network updating is possible for at least one case. *If there is a single node in the output layer of the network*, let $k = 1$ and define

$$w^{\dagger}_{j,l} = w_{k,j} w'_{j,l} = w_{1,j} w'_{j,l} \tag{2.45}$$

From (2.34) it follows that

$$(\bar{y}_1(n) - b_1(n)) = \sum_{j=1}^{N_1} \sum_{l=1}^{N_0} [K_1(n) K'_j(n) x_l(n)] w^{\dagger}_{j,l} + \sum_{j=1}^{N_1} [K_1(n) b'_j(n)] w_{1,j}. \tag{2.46}$$

This can be interpreted as an attempt to train a single linear layer with one output and $(N_0 \times N_1) + N_1$ inputs. In this case, there will be only N linearized training patterns. The system can be solved for $w_{1,j}$ and $w^{\dagger}_{j,l}$ and (2.45) can be used to solve for $w'_{j,l}$.

2.2.3.2 L Layer Network

As in the two layer case, the equations for layer wise weight updating in the L layer network come directly from the layer wise weight updating equations. Rearranging (2.41) and replacing k with 1 we get,

$$\bar{y}_1^{(L)}(n) - b_1^{(L)}(n) = \sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \cdots \sum_{j_{L-l-1}=1}^{N_2} \sum_{j_{L-l}=1}^{N_1} \sum_{j_{L-l+1}=1}^{N_0} \Big(K_1^{(L)}(n) K_{j_1}^{(L-1)}(n) \cdot \quad (2.47)$$

$$K_{j_{L-l}}^{(l)}(n) y_{j_{L-l+1}}^{(l-1)}(n) \Big) w_{1,j_1}^{(L)} w_{j_1,j_2}^{(L-1)} \cdots w_{j_{L-l-1},j_{L-l}}^{(1)} w_{j_{L-l},j_{L-l+1}}^{(0)}$$

$$+ \sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \cdots \sum_{j_{L-l-1}=1}^{N_2} \sum_{j_{L-l}=1}^{N_1} \Big(K_1^{(L)}(n) K_{j_1}^{(L-1)}(n) \cdots$$

$$b_{j_{L-l}}^{(0)} \Big) w_{1,j_1}^{(L)} w_{j_1,j_2}^{(L-1)} \cdots w_{j_{L-l-1},j_{L-l}}^{(1)} + \cdots$$

$$+ \sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \Big(K_1^{(L)}(n) K_{j_1}^{(L-1)}(n) b_{j_2}^{L-2} \Big) w_{1,j_1}^{(L)} w_{j_1,j_2}^{(L-1)}$$

$$+ \sum_{j_1=1}^{N_{L-1}} \Big(K_1^{(L)}(n) b_{j_1}^{L-1} \Big) w_{1,j_1}^{(L)}$$

This can be redefined as

$$\bar{y}_1^{(L)}(n) - b_1^{(L)}(n) = \quad (2.48)$$

$$\sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \cdots \sum_{j_{L-l-1}=1}^{N_2} \sum_{j_{L-l}=1}^{N_1} \sum_{j_{L-l+1}=1}^{N_0} \Big(K_1^{(L)}(n) K_{j_1}^{(L-1)}(n) \cdots$$

$$K_{j_{L-l}}^{(1)}(n) y_{j_{L-l+1}}^{(0)}(n) \Big) w_{j_1,j_2,\ldots,j_{L-l-1},j_{L-l},j_{L-l+1}}^{(L\dagger)}$$

$$+ \sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \cdots \sum_{j_{L-l-1}=1}^{N_2} \sum_{j_{L-l}=1}^{N_1} \Big(K_1^{(L)}(n) K_{j_1}^{(L-1)}(n) \cdots b_{j_{L-l}}^{(0)} \Big) w_{j_1,j_2,\ldots,j_{L-l-1},j_{L-l}}^{(L\dagger)}$$

$$+ \cdots$$

$$+ \sum_{j_1=1}^{N_{L-1}} \sum_{j_2=1}^{N_{L-2}} \Big(K_1^{(L)}(n) K_{j_1}^{(L-1)}(n) b_{j_2}^{L-2} \Big) w_{j_1,j_2}^{(L\dagger)}$$

$$+ \sum_{j_1=1}^{N_{L-1}} \Big(K_1^{(L)}(n) b_{j_1}^{L-1} \Big) w_{1,j_1}^{(L)}$$

This can be interpreted as a one output, $[N_L + (N_L \times N_{L-1}) + (N_L \times N_{L-1} \times N_{L-2})$ $+(N_L \times N_{L-1} \times \ldots \times N_1 \times N_0)]$ input linear system. The weights $w_{j_1,j_2,\ldots,j_{L-l-1},j_{L-l},j_{L-l+1}}^{(L\dagger)}$, $w_{j_{11},j_2,\ldots,j_{L-l-1},j_{L-l}}^{(L\dagger)}$, ..., $w_{j_1,j_2}^{(L\dagger)}$, $w_{1,j_1}^{(L)}$ can be solved for and used to determine the system weights. In the previous secion we saw that as the number of layers increased, the dimension of the inputs increased. Here as the number of layers increases, the

dimension of the weights increases.

2.3 Solving the Linearized Equations

2.3.1 Solution by MIL-WRLS

In principle, once the linearization is achieved at iteration i and pattern n, any least mean square type algorithm can be employed to update the weight estimates. The A-S method uses the conventional MIL-WRLS algorithm. In this case, neglecting any error weighting, MIL-WRLS takes the form of the two recursions (written for node k in the top layer) [5, Ch.5],

$$\boldsymbol{P}(i,n) = \boldsymbol{I} - \frac{\boldsymbol{P}(i,n-1)\bar{\boldsymbol{y}}'_k(n)[\bar{\boldsymbol{y}}'_k(n)]^T\boldsymbol{P}(i,n-1)}{1+[\bar{\boldsymbol{y}}'_k(n)]^T\boldsymbol{P}(i,n-1)\bar{\boldsymbol{y}}'_k(n)} \tag{2.49}$$

$$\boldsymbol{w}_k(i,n) = \boldsymbol{w}_k(i,n-1) + \boldsymbol{P}(n)\bar{\boldsymbol{y}}'_k(n)[\bar{t}_k(n) - \bar{y}_k(n)]. \tag{2.50}$$

$\boldsymbol{w}_k(i,n)$ is the estimate of the weights $\boldsymbol{w}_k$ following pattern n in the i^{th} iteration through the training data, and $\boldsymbol{P}^{-1}(i,n)$ is the covariance matrix at the same "time" in the process,

$$\boldsymbol{P}^{-1}(i,n) \stackrel{\text{def}}{=} \sum_{j=1}^{n} \bar{\boldsymbol{y}}'_k(j)[\bar{\boldsymbol{y}}'_k(j)]^T. \tag{2.51}$$

Note that $\boldsymbol{w}_k(i,0) \stackrel{\text{def}}{=} \boldsymbol{w}_k(i-1,N)$ and similarly for the covariance matrix. This presents the question of how $\boldsymbol{w}_k(0,0)$ and $\boldsymbol{P}(0,0)$ should be initialized. The inverse covariance matrix contains theoretically infinite values at the outset and a proper initialization for the weights is practically not known (this means that the initial linearizations of the training data are based on potentially very bad weight estimates). This issue will be addressed further below. Also, it is clear that this solution, as written, will continue to "accumulate" past linearized sets of data which might, in fact, be linearized around very poor weight estimates. Therefore, the A-S algorithm

includes a "forgetting factor" [5] in the MIL-WRLS recursions. This is equivalent to using a weighted error criterion with time varying (exponentially decaying) weights. This can make convergence slow if the forgetting factor is large. If the forgetting factor is small, then past values are forgotten more quickly, but the algorithm may have convergence problems. We will also comment further on this issue below.

We have found that the choice of conventional MIL-WRLS as a solution method seriously impairs the ability of this linearization method to converge on a proper set of network weights. As an alternative, therefore, we suggest the method presented in the following section.

2.3.2 Solution by QR-WRLS

In order to improve convergence the algorithm developed above can be implemented using QR-WRLS [4, 7, 28]. This algorithm has distinct advantages over conventional MIL-WRLS. First, the QR-WRLS algorithm does not suffer from initialization problems noted above for MIL-WRLS. It is also robust numerically, as no matrix inversions are done. This should prove to be very beneficial for different algorithms currently implemented with MIL-WRLS. Allred [29] made a study of various FNN training algorithms for the U.S. Navy and rejected all which used MIL-WRLS, because, in his words, "Classical techniques such as Newton's technique or regression analysis (which invert large $N \times N$ matrices) can be ill-conditioned and fail. ... Although these approaches may be interesting, we reject them in favor of more robust techniques. The last thing one wants is to have a software product which fails in a production environment." In addition to improved numerical properties, QR-WRLS also permits the inclusion of several very flexible "forgetting" strategies. To illustrate the operation of the algorithm, it is sufficient to consider the estimation of weights $\boldsymbol{w}_k$ in the output layer of the two layer network. All notation is consistent with that used above.

In effect, the linearization technique described above reduces the problem at the i^{th} iteration through the training patterns to one of finding the least square error solution of the overdetermined system of equations

$$\begin{bmatrix} (\bar{\boldsymbol{y}}_k'(1))^T & \rightarrow \\ (\bar{\boldsymbol{y}}_k'(2))^T & \rightarrow \\ \vdots & \vdots \\ (\bar{\boldsymbol{y}}_k'(N))^T & \rightarrow \end{bmatrix} \boldsymbol{w}_k(i) = \begin{bmatrix} \bar{t}_k(1) \\ \bar{t}_k(2) \\ \vdots \\ \bar{t}_k(N) \end{bmatrix}. \tag{2.52}$$

The QR-WRLS method is based upon transforming this system into an upper triangular system by applying a series of orthonormal operators (Givens rotations). The resulting system is

$$\begin{bmatrix} \boldsymbol{T}(i,N) \\ \hline \mathbf{0}_{(N-N_1)\times N_1} \end{bmatrix} \boldsymbol{w}_k(i) = \begin{bmatrix} \boldsymbol{d}_1(i,N) \\ \hline \boldsymbol{d}_2(i,N) \end{bmatrix} \tag{2.53}$$

where the matrix $\boldsymbol{T}(i,N)$ is $N_1 \times N_1$ upper triangular and $\mathbf{0}_{j\times k}$ denotes the $j \times k$ zero matrix. The solution for $\boldsymbol{w}_k(i)$ is easily obtained by back-substitution. A recursive version of the solution is also possible. The recursive algorithm is shown in Figure 2.4. For details the reader is referred to [4, 28].

For discussion of further benefits of the decomposition algorithm, it is useful to view the $\boldsymbol{A}$ matrix, defined in Figure 2.4, as four partitions. Following the rotation of the n^{th} equation, in Step 2, for example,

$$\boldsymbol{A} = \left[\begin{array}{c|c} \boldsymbol{T}(i,n) & \boldsymbol{d}_1(i,n) \\ \hline \mathbf{0}_{1\times N_1} & d_2(i,n) \end{array}\right]. \tag{2.55}$$

As is the case with the A-S method, a forgetting factor must be employed to gradually reduce the effects of earlier linearizations. This is very easily accomplished in the

Figure 2.4: Weight estimation using recursive QR-WRLS

WEIGHT ESTIMATION USING RECURSIVE QR-WRLS

Initialization: Initialize an $(N_1 + 1) \times (N_1 + 1)$ working matrix, say $\boldsymbol{A}$, to a null matrix.

Recursion: For $i = 1, 2, \ldots$ (iteration); and, For $n = 1, 2, \ldots, N$ (pattern),

1. Enter the next equation into the bottom row of $\boldsymbol{A}$,

$$\left[\; [\bar{\boldsymbol{y}}'_k(n)]^T \; \middle| \; \bar{t}_k(n) \; \right]. \tag{2.54}$$

2. "Rotate" the new equation into the system using

$$\begin{aligned} A'_{mk} &= A_{mk}\sigma + A_{N_1+1,k}\tau\mathcal{S} \\ A'_{N_1+1,k} &= -A_{mk}\tau\mathcal{S} + A_{N_1+1,k}\sigma\mathcal{S} \end{aligned}$$

for $k = m, m+1, \ldots, N_1 + 1$ and $m = 1, \ldots, N_1$; where $\sigma = A_{mm}/\rho$, $\tau = A_{N_1+1,m}/\rho$, $\rho = (A^2_{mm} + A^2_{N_1+1,m})^{1/2}$, $\mathcal{S}$ is unity (useful later), and $A_{mk}(A'_{mk})$ is the m, k element of $\boldsymbol{A}$ pre- (post-)rotation. No other elements of $\boldsymbol{A}$ are affected.

3. Solve for the least square estimate of the weights $\boldsymbol{w}_k$ if desired. (Solution after the n^{th} pattern will produce what has been called $\boldsymbol{w}_k(i, n)$ in the text, and $\boldsymbol{w}_k(i, N) = \boldsymbol{w}_k(i)$.)

4. If $n < N$, increment n. Otherwise check convergence criterion and increment i and reset n if not met.

Termination: Stop when some convergence criterion is met.

QR-WRLS algorithm by simply multiplying the top N_1 rows of the matrix $\boldsymbol{A}$ (matrix $\boldsymbol{T}(i,n)$ and vector $\boldsymbol{d}_1(i,n)$) by a factor $\beta < 1$ prior to the rotation of the $n+1^{st}$ pattern equation. In this context, both the forgetting factor and the frequency of weight updates can be varied. In addition to exponential forgetting factors, equations can be "rotated out" of the matrix. This is done by changing $\mathcal{S}$ in Figure 2.4 to -1 and rotating in the equation to eliminate. Thus, for example, only the last $Q > N_1$ equations can be used to calculate the weight updates by sequentially removing equation $n-Q+1$ prior to inclusion of equation n. This procedure effects a sliding window over which the estimates are computed. Another forgetting method useful for FNNs is possible because no initialization of the updating equations is necessary. Because there are no initialization problems, the system can be re-initialized at any step, thus completely "forgetting" the past linearized values. One method that has worked well in our simulations is to re- initialize the $\boldsymbol{A}$ matrix after every iteration. These and a number of other flexible forgetting strategies made possible by this algorithm may prove very useful in the training of FNNs [28]. In addition to new forgetting factors, using the QR-WRLS implementation also allows the frequency of updating of the weights to vary. As with conventional MIL-WRLS, the weights can be updated every time a new linearization is used[6].

The theoretical results above, along with those in Section 2, can be combined to form a learning algorithm for FNNs. For the node-wise weight updating this is done as follows. First, the weights of the network are initialized. This is done randomly, each weight being selected from a uniform distribution over the set [-1,1]. Once the initial weights are chosen, the weight updating can begin. First, a training pattern is input to the system. Because the weights are not updated until all the training patterns have been used, convergence does not depend on the order in which

[6]This can be as often as every pattern, or at the end of each iteration through the patterns as has been our convention.

the training patterns are used. Given a training pattern, the algorithm calculates linearized training patterns for the last layer nodes and these are rotated into the corresponding $\boldsymbol{A}$ matrix. Each node has a separate "$\boldsymbol{A}$" matrix. The target outputs of the layer below are calculated next using back-propagation, see (2.16). The $\boldsymbol{A}$ matrices for the first layer are then updated. A new training pattern is then used to calculate a new set of linearized inputs and outputs. This is repeated until all the training patterns have been used. The $\boldsymbol{A}$ matrices are then used to calculate updated weights. This continues until the network converges to a solution or a specified maximum number of iterations has been reached. By definition the solution is said to have converged when the change of the norm of the vector of all the weights is below a threshold. The algorithm for the layer-wise weight updating is similar. Weight updating is exactly the same for the output layer nodes. The layers below have one $\boldsymbol{A}$ matrix each. The linearized training patterns are calculated for each layer and rotated into the corresponding matrix. After all the training patterns have been rotated in, the new weights are calculated and the process is repeated. As before the algorithm stops when a solution is reached or a specified maximum number of iterations is reached. As with other training algorithms for FNNs, this algorithm may not converge to the weights corresponding to the *global* minimum of the function of E. Also, although the algorithm approximates a gradient system, because it is not a gradient system, there is no guarantee that the algorithm will converge to any solution.

Chapter 3

Data Reduction Algorithm

3.1 Introduction

This chapter describes a method of reducing the number of training patterns used at each iteration. Each linearized training pattern that is rotated into $\boldsymbol{T}(i, N)$ and $\boldsymbol{d}_1(i, N)$ will affect both the magnitude and direction of the vector solution $\boldsymbol{w}(i)$. Some training patterns will have a relatively large effect on the solution and others a relatively small effect. The goal is to be able to determine which training patterns will have a very small effect on the resulting weights and avoid the rotation of these into the system of equations, thereby reducing the number of computations at this iteration and making training more efficient.

3.2 Data Reduction Algorithm

Simulation studies have revealed a very important phenomenon which occurs when the training algorithm of the previous chapter is employed. As the training proceeds, a few of the linearized training patterns become dominant, and the weight change is largely dependent on these training patterns. Although the *original* training patterns do not change, the *linearized* training patterns are dependent on the present weights and so the *linearized* training patterns generally change at each iteration. For this

reason, at one iteration a linearized training pattern may have a large effect on the weight change while at another iteration, after the weights have changed, the linearized training pattern may have a negligible effect. All linearized weight updating algorithms presented in Chapter 2 in effect reduce the network to a linear network with one output. Hence in this section we will consider one node with M inputs and one output, using the notation introduced for QR-WRLS.

The fact that a few of the linearized training patterns become dominant during the training is due to the non-linearity in the output. Simply put, it is usual for the linearized training patterns to decrease in magnitude as the training progresses, and so have a decreasing effect on the resulting vector $\boldsymbol{w}(i)$. Let us now consider this effect in greater detail. The non-linearity used in the FNN is a sigmoid. As a consequence, as the input to the nonlinearity of the node approaches $\pm\infty$ the output approaches a constant. The derivative of the output with respect to the input, which is always positive, is largest when the input is zero, and goes to zero as the absolute value of the input becomes large. This derivative is important in determining the effect of a linearized training pattern on the result because the magnitude of the linearized training pattern is proportional to this derivative. The derivative, in turn, is dependent on the input u. Finally, u is dependent on the weights and on the non-linear training pattern.

As the weights change during the training process, we can view the changing effect that a specific linearized training pattern has on the solution relative to itself, or relative to the other linearized training patterns. First, let us view the changing effect of a linearized training pattern relative to itself. As training progresses, it is usual for the weights to increase. This increase in weights usually causes the input u to increase. An increase in u causes the derivative at successive iterations to decrease, and the magnitude of the linearized training pattern usually decreases. This can be easily seen from (2.9) and (2.13). The linearized training pattern at the next iterations

will have a smaller magnitude and thus have a smaller effect on the solution weights.

Now consider a specific linearized training pattern relative to other linearized training patterns. It is usual for all the linearized training patterns to decrease in magnitude and thus have a decreasing effect on the solution weights. However, because the change of weights does not affect the input u corresponding to each of the training patterns equally, the derivative of the output with respect to the input will not change equally for all the training patterns. As the derivative for some training pattern becomes very small the effect of this linearized training pattern becomes negligible relative to the others.

Another way of explaining this process involves the use of error surfaces. Each training pattern has an associated error surface that is a function of the system weights. As shown in (1.6), the total error surface is the sum of the error surfaces associated with each of the individual training patterns. A training pattern with an associated error surface that is almost flat at the present weights does not contribute significantly to the derivative of the total error surface, and consequently does not contribute significantly to the determination of the gradient direction. The effect of rotating a training pattern with a small corresponding derivative, one whose error surface is almost flat at the present weights, into the $\boldsymbol{T}(i, N)$ matrix is smaller than the effect of another training pattern with a large derivative, or an error surface that is not as flat.

To determine whether some of the training patterns can be ignored at the present iteration without affecting the solution significantly, two issues must be addressed: how to determine if a training pattern has an effect small enough to be excluded at the present iteration, and how to do so in a computationally inexpensive way. The "effect" of each training pattern will be measured by the magnitude of the change of the solution weights due to this training pattern.

Assume that iteration i through the training patterns has been completed. Thus,

$\boldsymbol{w}(i)$ has been computed, and the system weights have been changed in the direction of $\boldsymbol{w}(i)$. New linearized training patterns are then calculated using these new weights. We now need to decide which training patterns to use in the calculation of $\boldsymbol{w}(i+1)$.

The following matrix equation is the result of the i^{th} iteration :

$$\boldsymbol{T}(i,N)\boldsymbol{w}(i) = \boldsymbol{d}_1(i,N) \tag{3.1}$$

Suppose we, use QR-WRLS to rotate the first new linearized training pattern into (3.1). This results in a new matrix equation, say,

$$\boldsymbol{T}'(i,N)\boldsymbol{w}'(i) = \boldsymbol{d}'_1(i,N) \tag{3.2}$$

Let

$$\begin{aligned} \delta\boldsymbol{w}(i) &= \boldsymbol{w}'(i) - \boldsymbol{w}(i) \\ \delta\boldsymbol{T}(i,N) &= \boldsymbol{T}'(i,N) - \boldsymbol{T}(i,N) \\ \delta\boldsymbol{d}_1(i,N) &= \boldsymbol{d}'_1(i,N) - \boldsymbol{d}_1(i,N) \end{aligned} \tag{3.3}$$

Thus $\| \delta\boldsymbol{w}(i) \|$ is the magnitude of the weight change due to this training pattern at this iteration, where, for the moment, $\| \cdot \|$ indicates any valid norm. To simplify notation

$$\Delta(\boldsymbol{w}(i)) = \frac{\| \delta\boldsymbol{w}(i) \|}{\| \boldsymbol{w}(i) \|} \tag{3.4}$$

will be used in comparing the magnitudes of the weight changes due to the training patterns. The same procedure of rotating one training pattern into (3.1) is repeated in order to calculate $\Delta(\boldsymbol{w}(i))$ separately for each of the linearized training patterns. We now have a number for comparing the effect of each training pattern. However, it is immediately apparent that in order to calculate $\Delta(\boldsymbol{w}(i))$ for each training pattern using any norm, each training pattern must be rotated separately into the system

(3.1). This is the same number of computations needed to calculate new weights using all the training patterns. Consequently, direct computation of $\Delta(\boldsymbol{w}(i))$ as a means of checking for usefulness of the given training pattern offers no computational advantage. We therefore seek an approximation to $\Delta(\boldsymbol{w}(i))$, say $\hat{\Delta}(\boldsymbol{w}(i))$, which is

1. much less computationally expensive to compute than $\Delta(\boldsymbol{w}(i))$.

Additionally, it will be desirable for $\hat{\Delta}(\boldsymbol{w}(i))$ to have two further properties:

2. Order preservation. If $\Delta^{(n)}(\boldsymbol{w}(i))$ indicates the normalized change in $\boldsymbol{w}(i)$ due to training pattern n, and $\Delta^{(n)}(\boldsymbol{w}(i)) < \Delta^{(m)}(\boldsymbol{w}(i))$, then it should be true that $\hat{\Delta}^{(n)}(\boldsymbol{w}(i)) < \hat{\Delta}^{(m)}(\boldsymbol{w}(i))$.

3. $\hat{\Delta}^{(n)}(\boldsymbol{w}(i)) \rightarrow 0$ as $\Delta^{(n)}(\boldsymbol{w}(i)) \rightarrow 0$.

Properties 2 and 3 assure that the approximation will "track" the true change at small values (where the information is most important) while at the same time preserving the order of relative changes among the patterns.

Now, from (3.2) and (3.3)

$$[\boldsymbol{T}(i,N) + \delta\boldsymbol{T}(i,N)]\,[\boldsymbol{w}(i) + \delta\boldsymbol{w}(i)] = \boldsymbol{d}_1(i,N) + \delta\boldsymbol{d}_1(i,N) \tag{3.5}$$

Using (3.1), (3.5) becomes

$$\delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) + \boldsymbol{T}(i,N)\delta\boldsymbol{w}(i) + \delta\boldsymbol{T}(i,N)\delta\boldsymbol{w}(i) = \delta\boldsymbol{d}_1(i,N). \tag{3.6}$$

Factoring $\delta\boldsymbol{w}(i)$,

$$[\boldsymbol{T}(i,N) + \delta\boldsymbol{T}(i,N)]\,\delta\boldsymbol{w}(i) = [\delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i)] \tag{3.7}$$

After some manipulation, we obtain

$$\delta \boldsymbol{w}(i) = \boldsymbol{T}^{-1}(i,N)\left[[\delta \boldsymbol{d}_1(i,N) - \delta \boldsymbol{T}(i,N)\boldsymbol{w}(i)] - \delta \boldsymbol{T}(i,N)\delta \boldsymbol{w}(i)\right]. \tag{3.8}$$

Taking any valid norm gives the upper bound

$$\| \delta \boldsymbol{w}(i) \| \leq \| \boldsymbol{T}^{-1}(i,N) \| \left[\| \delta \boldsymbol{d}_1(i,N) - \delta \boldsymbol{T}(i,N)\boldsymbol{w}(i) \| + \| \delta \boldsymbol{T}(i,N) \| \| \delta \boldsymbol{w}(i) \| \right], \tag{3.9}$$

which, after some algebra, yields,

$$\| \delta \boldsymbol{w}(i) \| \leq \frac{\| \boldsymbol{T}^{-1}(i,N) \| \cdot \| \delta \boldsymbol{d}_1(i,N) - \delta \boldsymbol{T}(i,N)\boldsymbol{w}(i) \|}{1 - \| \boldsymbol{T}^{-1}(i,N) \| \cdot \| \delta \boldsymbol{T}(i,N) \|}. \tag{3.10}$$

It is our objective to produce an *approximation* to $\| \delta \boldsymbol{w}(i) \|$ or $\Delta(\boldsymbol{w}(i))$ rather than a bound. A pivotal consideration in doing so is the conditioning of the matrix $\boldsymbol{T}(i,N)$. We digress momentarily to consider this issue. Proper conditioning of $\boldsymbol{T}(i,N)$ and $\boldsymbol{T}(i,N) + \delta \boldsymbol{T}(i,N)$ mean that

$$\begin{aligned} \| \boldsymbol{T}(i,N) \| \cdot \| \boldsymbol{T}^{-1}(i,N) \| &\approx 1 \\ \| \boldsymbol{T}(i,N) + \delta \boldsymbol{T}(i,N) \| \cdot \| \boldsymbol{T}^{-1}(i,N) + \delta \boldsymbol{T}(i,N) \| &\approx 1 \end{aligned} \tag{3.11}$$

Geometrically this implies that neither of the products

$$\boldsymbol{T}(i,N)\boldsymbol{w}(i) \tag{3.12}$$

$$[\boldsymbol{T}(i,N) + \delta \boldsymbol{T}(i,N)]\boldsymbol{w}(i) \tag{3.13}$$

is dependent on the direction of $\boldsymbol{w}(i)$. The ability to discount the direction of $\boldsymbol{w}(i)$ will be central to the development of a useful approximation. A sufficient condition for these matrices to be well conditioned is for the features to be independent and

identically distributed (i.i.d.). This is because $\boldsymbol{T}(i,N)$ is the Cholesky factor of the "temporal" covariance matrix whose expectation is diagonal under proper ergodicity assumptions. The i.i.d. feature model is not unrealistic in many problems.

Returning to the bound in (3.10), we invoke the fact that if the $\boldsymbol{T}^{-1}(i,N)$ matrix, and equivalently the $\boldsymbol{T}(i,N)$ matrix, is well-conditioned, then we can replace the inequality by an approximation,

$$\| \delta \boldsymbol{w}(i) \| \approx \frac{\| \boldsymbol{T}^{-1}(i,N) \| \cdot \| \delta \boldsymbol{d}_1(i,N) - \delta \boldsymbol{T}(i,N) \boldsymbol{w}(i) \|}{1 - \| \boldsymbol{T}^{-1}(i,N) \| \cdot \| \delta \boldsymbol{T}(i,N) \|}. \tag{3.14}$$

This is possible because the Cauchy-Schwartz inequality can be approximated by an equality under the assumed conditions. While this approximation can be shown to have the second and third features mentioned above, it is too expensive computationally to be useful in speeding up the algorithm. It would require more computations to calculate this approximation than to use all the training patterns for the weight updating. In order to reduce the number of computations, substitutions for some of the factors in this approximation will be made. The first substitution introduced will be for $\| \boldsymbol{T}^{-1}(N) \|$ (see [30]),

$$\frac{\| \boldsymbol{w}(i) \|}{\| \boldsymbol{d}_1(i,N) \|} \leq \| \boldsymbol{T}^{-1}(i,N) \| . \tag{3.15}$$

It is easily shown that under the same well-conditioning assumption, the inequality is also a good approximation, so,

$$\frac{\| \boldsymbol{w}(i) \|}{\| \boldsymbol{d}_1(i,N) \|} \approx \| \boldsymbol{T}^{-1}(i,N) \| . \tag{3.16}$$

Making this substitution for $\| \boldsymbol{T}^{-1}(i,N) \|$ on the right side of (3.10), we obtain

$$\| \delta \boldsymbol{w}(i) \| \approx \frac{\frac{\| \boldsymbol{w}(i) \|}{\| \boldsymbol{d}_1(i,N) \|} \| \delta \boldsymbol{d}_1(i,N) - \delta \boldsymbol{T}(i,N) \boldsymbol{w}(i) \|}{1 - \frac{\| \boldsymbol{w}(i) \|}{\| \boldsymbol{d}_1(i,N) \|} \| \delta \boldsymbol{T}(i,N) \|}. \tag{3.17}$$

Thus

$$\Delta(\boldsymbol{w}(i)) = \frac{\| \delta\boldsymbol{w}(i) \|}{\| \boldsymbol{w}(i) \|} \approx \frac{\| \delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|}{\| \boldsymbol{d}_1(i,N) \| - \| \boldsymbol{w}(i) \| \| \delta\boldsymbol{T}(i,N) \|} \stackrel{\text{def}}{=} \hat{\Delta}(\boldsymbol{w}(i)) \qquad (3.18)$$

Let us now show that under fairly general conditions, (3.18) is a good approximation. First we demonstrate order preservation. Let $\| \delta^{(n)}\boldsymbol{w}(i) \|$ and $\| \delta^{(m)}\boldsymbol{w}(i) \|$ be the weight changes due to two different training patterns, n and m. Let us suppose that

$$\| \delta^{(n)}\boldsymbol{w}(i) \| < \| \delta^{(m)}\boldsymbol{w}(i) \| . \qquad (3.19)$$

Therefore

$$\| \boldsymbol{T}(i,N) \| \cdot \| \delta^{(n)}\boldsymbol{w}(i) \| < \| \boldsymbol{T}(i,N) \| \cdot \| \delta^{(m)}\boldsymbol{w}(i) \| . \qquad (3.20)$$

Let us assume that, for all training patterns, $\delta\boldsymbol{T}(i,N)$ is small compared with $\boldsymbol{T}(i,N)$ so that

$$\| \boldsymbol{T}(i,N) \| \approx \| \boldsymbol{T}(i,N) + \delta\boldsymbol{T}(i,N) \| \qquad (3.21)$$

and

$$\frac{\| \boldsymbol{d}_1(i,N) \|}{\| \boldsymbol{w}(i) \|} >> \| \delta\boldsymbol{T}(i,N) \| . \qquad (3.22)$$

With assumption (3.21), we can write (3.20) as

$$\| \boldsymbol{T}(i,N) + \delta^{(n)}\boldsymbol{T}(i,N) \| \cdot \| \delta^{(n)}\boldsymbol{w}(i) \| < \| \boldsymbol{T}(i,N) + \delta^{(m)}\boldsymbol{T}(i,N) \| \cdot \| \delta^{(m)}\boldsymbol{w}(i) \| . \qquad (3.23)$$

Where $\delta^{(n)}\boldsymbol{T}(i,N)$ is the change in $\boldsymbol{T}(i,N)$ due to pattern n. Because $\boldsymbol{T}(i,N) + \delta^{(n)}\boldsymbol{T}(i,N)$ and $\boldsymbol{T}(i,N) + \delta^{(m)}\boldsymbol{T}(i,N)$ are assumed well conditioned, (3.23) can be written

$$\| [\boldsymbol{T}(i,N) + \delta^{(n)}\boldsymbol{T}(i,N)]\delta^{(n)}\boldsymbol{w}(i) \| < \| [\boldsymbol{T}(i,N) + \delta^{(m)}\boldsymbol{T}(i,N)]\delta^{(m)}\boldsymbol{w}(i) \| . \qquad (3.24)$$

Now using condition (3.22),

$$\frac{\| [\boldsymbol{T}(i,N) + \delta^{(n)}\boldsymbol{T}(i,N)]\delta^{(n)}\boldsymbol{w}(i) \|}{\frac{\|\boldsymbol{d}_1(i,N)\|}{\|\boldsymbol{w}(i)\|} - \| \delta^{(n)}\boldsymbol{T}(i,N) \|} < \frac{\| [\boldsymbol{T}(i,N) + \delta^{(m)}\boldsymbol{T}(i,N)]\delta^{(m)}\boldsymbol{w}(i) \|}{\frac{\|\boldsymbol{d}_1(i,N)\|}{\|\boldsymbol{w}(i)\|} - \| \delta^{(m)}\boldsymbol{T}(i,N) \|} \tag{3.25}$$

From (3.7), (3.25) is equivalent to the inequality

$$\frac{\| \delta\boldsymbol{d}_1(i,N) - \delta^{(n)}\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|}{\frac{\|\boldsymbol{d}_1(i,N)\|}{\|\boldsymbol{w}(i)\|} - \| \delta^{(n)}\boldsymbol{T}(i,N) \|} < \frac{\| \delta\boldsymbol{d}_1(i,N) - \delta^{(m)}\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|}{\frac{\|\boldsymbol{d}_1(i,N)\|}{\|\boldsymbol{w}(i)\|} - \| \delta^{(m)}\boldsymbol{T}(i,N) \|} \tag{3.26}$$

or

$$\frac{\| \delta\boldsymbol{d}_1(i,N) - \delta^{(n)}\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|}{\| \boldsymbol{d}_1(i,N) \| - \| \boldsymbol{w}(i) \| \cdot \| \delta^{(n)}\boldsymbol{T}(i,N) \|} < \frac{\| \delta\boldsymbol{d}_1(i,N) - \delta^{(m)}\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|}{\| \boldsymbol{d}_1(i,N) \| - \| \boldsymbol{w}(i) \| \cdot \| \delta^{(m)}\boldsymbol{T}(i,N) \|} \tag{3.27}$$

Hence we have shown that the ordering of weight effects (3.19) results in

$$\hat{\Delta}^{(n)}(\boldsymbol{w}(i)) < \hat{\Delta}^{(m)}(\boldsymbol{w}(i)) \tag{3.28}$$

and ordering is preserved. It is of interest to note that starting with (3.27), we can derive (3.19) with the assumptions in (3.21) and (3.22), without the need for the well conditioning of the matrices.

Next we investigate the behavior of $\hat{\Delta}(\boldsymbol{w}(i))$ when $\| \delta\boldsymbol{w}(i) \|$ is small. In reducing the number of training patterns to be used in the updating process, we want to detect the training patterns with very small effects, and avoid the computations necessary to include them. For any valid norm,

$$\| \delta\boldsymbol{w}(i) \| = 0 \Rightarrow \delta\boldsymbol{w}(i) = 0 \tag{3.29}$$

In this case it is easily shown using (3.2) and (3.3) that

$$\| \delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \| = 0. \tag{3.30}$$

It follows immediately that $\hat{\Delta}(\boldsymbol{w}(i)) = 0$. Consequently, the "tracking to zero" property is established.

Thus it can be seen that if the $\boldsymbol{T}(i,N)$ matrices are well conditioned, then the approximation is good in the sense of properties 2 and 3. If the $\boldsymbol{T}(i,N)$ matrix is not well-conditioned, then two errors can occur. First, a training pattern with a small effect may be included in the weight updates. This "false alarm" is not a serious problem because including additional training patterns is not detrimental to the weight estimation. Further, the numerator in the approximation is greatest when the matrices are well conditioned, thus,

$$\| [\boldsymbol{T}(i,N) + \delta\boldsymbol{T}(i,N)]\delta\boldsymbol{w}(i) \| \leq \| \boldsymbol{T}(i,N) + \delta\boldsymbol{T}(i,N) \| \cdot \| \delta\boldsymbol{w}(i) \| . \qquad (3.31)$$

If the weight changes are small, then the approximation will be smaller than if the matrices were well-conditioned, and the chance of a false alarm is small. The second error that can occur is the improper omission of a training pattern. A "miss" means that the left side of (3.31) is much smaller than the right, implying the matrix $\boldsymbol{T}(i,N) + \delta\boldsymbol{T}(i,N)$ is not well-conditioned. This ill conditioning occurs when there is a feature that is always "zero", so that a "zero" row appears in $\boldsymbol{T}(i,N)$, or if there is a poor scaling of the features, so that the diagonal elements of $\boldsymbol{T}(i,N)$ are not of comparable magnitudes. Either of these conditions implies a poorly designed experiment, and could be corrected by eliminating unuseful features or by scaling the features correctly.

Finally we must explore the computationally complexity of $\hat{\Delta}(\boldsymbol{w}(i))$. Although this approximation is less expensive than $\Delta(\boldsymbol{w}(i))$, computing it for each training pattern will again be more expensive than simply using all the training patterns in the updating equations. In order to achieve the desired reduced computational load, two more substitutions, one for $\| \delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|$ and for $\| \delta\boldsymbol{T}(i,N) \|$ will

be introduced. To reduce the number of computations needed to calculate $\boldsymbol{w}(i+1)$, the total number of computations needed to calculate the approximation $\hat{\Delta}(\boldsymbol{w}(i))$ for each training pattern and to use the chosen training patterns in the calculation of new weights, must be less than using all the training patterns to calculate new weights. Toward this end, let all the norms be the infinity norm. For a vector, this norm is the maximum of the absolute value of the elements, and the induced matrix norm is the maximum of the sums of the absolute values of the row components. To move to iteration $i+1$, $\| \boldsymbol{d}_1(i,N) \|_\infty$ and $\| \boldsymbol{w}(i) \|_\infty$ are determined from the results of the previous iteration. Because these quantities are independent of the linearized training patterns calculated at the iteration $i+1$, they need to be calculated only once. This means the approximation introduced in (3.16) is the same for all the training patterns.

This leaves $\| \delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|_\infty$ and $\| \delta\boldsymbol{T}(i,N) \|_\infty$ to be calculated for each linearized training pattern. If it were known which row of $\delta\boldsymbol{T}(i,N)$ be the largest, it would only be necessary to compute one row of $\boldsymbol{T}'(i,N)$ for each training pattern to get $\| \delta\boldsymbol{T}(i,N) \|_\infty$, and similarly for $\| \boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|_\infty$. In order to have fewer computations, it would be better, if possible, to use the same row to calculate $\| \delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|_\infty$ and $\| \delta\boldsymbol{T}(i,N) \|_\infty$. In this case, only one row of $\boldsymbol{T}'(i,N)$ would be calculated for each training pattern, and a savings in computations would result. For this reason, a judiciously selected single row will be used to approximate $\| \delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|_\infty$ and $\| \delta\boldsymbol{T}(i,N) \|_\infty$. Since $\boldsymbol{T}(i,N)$ and $\boldsymbol{T}'(i,N)$, and thus $\delta\boldsymbol{T}(i,N)$, are upper triangular, only the first row of $\delta\boldsymbol{T}(i,N)$ can have all non-zero elements. Consequently, a natural approximation for $\| \delta\boldsymbol{T}(i,N) \|_\infty$ is the sum of the absolute values of the first row elements. The norm of the first row is also a good approximation for $\| \delta\boldsymbol{d}_1(i,N) - \delta\boldsymbol{T}(i,N)\boldsymbol{w}(i) \|_\infty$. This is not only because it has more non-zero elements, but also because more of the weights are included in the computation of the row of $\delta\boldsymbol{T}(i,N)\boldsymbol{w}(i)$ actually used in

the computations. This means a greater directional component is included in the final computation. For example, if the last row of $\delta \boldsymbol{T}(i,N)$ were used, the contribution of $\delta \boldsymbol{T}(i,N)\boldsymbol{w}(i)$ would depend on only one weight. By using the first row, the employed row of $\delta \boldsymbol{T}(i,N)\boldsymbol{w}(i)$, and therefore $\| \delta \boldsymbol{d}_1(i,N) - \delta \boldsymbol{T}(i,N)\boldsymbol{w}(i) \|_\infty$, depend on all the weights.

These substitutions are necessary in order to have a reduction in the number of computations. Some consequences of using the substitutions can be explicitly observed. First, by approximating the infinity norm by one row, the approximation is smaller or equal to the true norm. If the true norm is the sum of the absolute values of the first row components, then the approximation is exact. If not, the approximation will be smaller than the true norm. This is true for both $\| \delta \boldsymbol{d}_1(i,N) - \delta \boldsymbol{T}(i,N)\boldsymbol{w}(i) \|_\infty$ and $\| \delta \boldsymbol{T}(i,N) \|_\infty$. Both of these approximations have the effect of making the weight change approximation smaller than the upper bound given in (3.10). The substitution for $\| \boldsymbol{T}^{-1}(i,N) \|_\infty$ used for (3.18) has the same effect. This means that the final approximation for $\| \delta \boldsymbol{w}(i) \|_\infty \; / \; \| \boldsymbol{w}(i) \|_\infty$ is bounded above by the right side of (3.10). Consequently, if the upper bound of the weight change is going to zero, the approximation will go to zero also. However, it is possible for the approximation to go to zero without the true weight change going to zero. Although this can occur theoretically, it has not proved to be a problem in implementation, as it does not occurr in any of the simulations reported in Chapter 4.

The practical test of the approximation to the weight change is its effectiveness as an indicator of the effect of the training pattern on the solution weights. Simulations were run in which the norm of the true weight change was compared with the approximation. The correlation coefficient between the norm of the true weight change and the approximation was computed for each iteration. It was typically in the range of 0.8 to 0.95. These results, along with the simulations run using the data reduction algorithm, indicate that the approximation is indeed good at estimating the weight

change of the linearized training pattern.

Chapter 4

Simulation Results

4.1 Introduction

This chapter is divided into three main sections. Each section corresponding to one of the three main contributions of this dissertation enumerated at the beginning of Chapter 1.

The first section presents the simulation results comparing BP, the A-S algorithm, and the QR-WRLS-based node-wise weight updating algorithm develped in this research. The principal purpose of this first group of simulations is to show the advantages of implementing linearized algorithms with QR-WRLS instead of MIL-WRLS. The simulation results for BP have been included as a reference.

Simulation results for BP, the node-wise, layer-wise and network-wise weight updating algorithms are presented in the next section. The node-wise and layer-wise algorithms are used for all the simulations, the network-wise algorithm is used for the simulations of a one output network.

The last section presents simulation studies for the data reduction scheme implemented for the layer-wise algorithm. BP results for the same studies are again presented as a reference.

4.2 Implementation Studies

QR-WRLS, presented in Chapter 2, has been shown to be slightly more computationally efficient than conventional MIL-WRLS in identification of linear systems [4]. The simulations presented here are designed to show the benefits of QR-WRLS for the non-linear FNN. The *theoretical* advantages are described in Chapter 2. These were confirmed in these simulations. One of these advantages is that the QR-WRLS equations need no initialization. With the MIL-WRLS implementation, the equations are initialized and the linearized training patterns are used to update the weight estimates. As mentioned in section 2.3.1, even though a forgetting factor is included, past training patterns are still incorporated in the computation of all weights. However, as the objective is to determine the gradient of the system at the *present* weights, including past training patterns will necessarily lead to an incorrect gradient estimate. The QR-WRLS equations on the other hand need no initialization. This allows the use of a forgetting factor if desired, or the linearized training patterns of the present iteration alone can be used to determine the gradient at the present weights. Even with simple networks and few training patterns, the difference between implementation using QR-WRLS and MIL-WRLS with a forgetting factor was apparent. With more complicated networks and many training patterns the error surface is more complicated and the difference in implementation is even greater. In this section the simulations were run with the QR-WRLS implementation using the same forgetting factor as the MIL-WRLS equations. This was done to simplify the comparisons. It will be shown in the next section that a smaller forgetting factor, or only using the training patterns of the present iteration, is better. This means that the results for the QR- WRLS implementation can be even better than those presented in this section.

Another significant advantage of QR-WRLS is the robust numerical properties. This was especially apparent in the ease of implementation. For a linear network, the training patterns are used once. With the FNN, the training patterns are presented

many times. As described in Chapter 3, the linearized training patterns tend to decrease in magnitude as the training progresses. This leads to the covariance matrix becoming very small, and its inverse, the $\boldsymbol{P}$ matrix of the MIL-WRLS equations, becoming very large. In the simulations this matrix tended to become numerically unstable. The QR-WRLS algorithm has no inversions so this is not a problem.

The simulations reported in this section compare three training strategies for an FNN. These are:

1. Conventional back-propagation, BP (no linearization).
2. The A-S algorithm (conventional MIL-WRLS with a forgetting factor).
3. Node-wise weight updating algorithm implemented with QR-WRLS with an exponential forgetting factor.

The linearizations performed in the A-S algorithm and the node-wise updating algorithms are equivalent, so the only differences between the last two algorithms is the implementation. Each of the three strategies above was used to train each of the following networks:

1. a 2-bit parity checker,
2. a 4-bit parity checker, and
3. a 4-bit bit counter.

The architectures for these three networks are illustrated in Figure 4.5.

The 2-bit parity checker (XOR) network has two inputs, two hidden layer nodes and one output node. An additional input is added at each layer whose value is always unity, to serve as a bias for each node. The output function $S(\cdot)$ is the sigmoid defined in (1.3). The initial weights were chosen as follows. Each weight in the network was selected randomly from a uniform distribution over the set $[-1, 1]$. This procedure

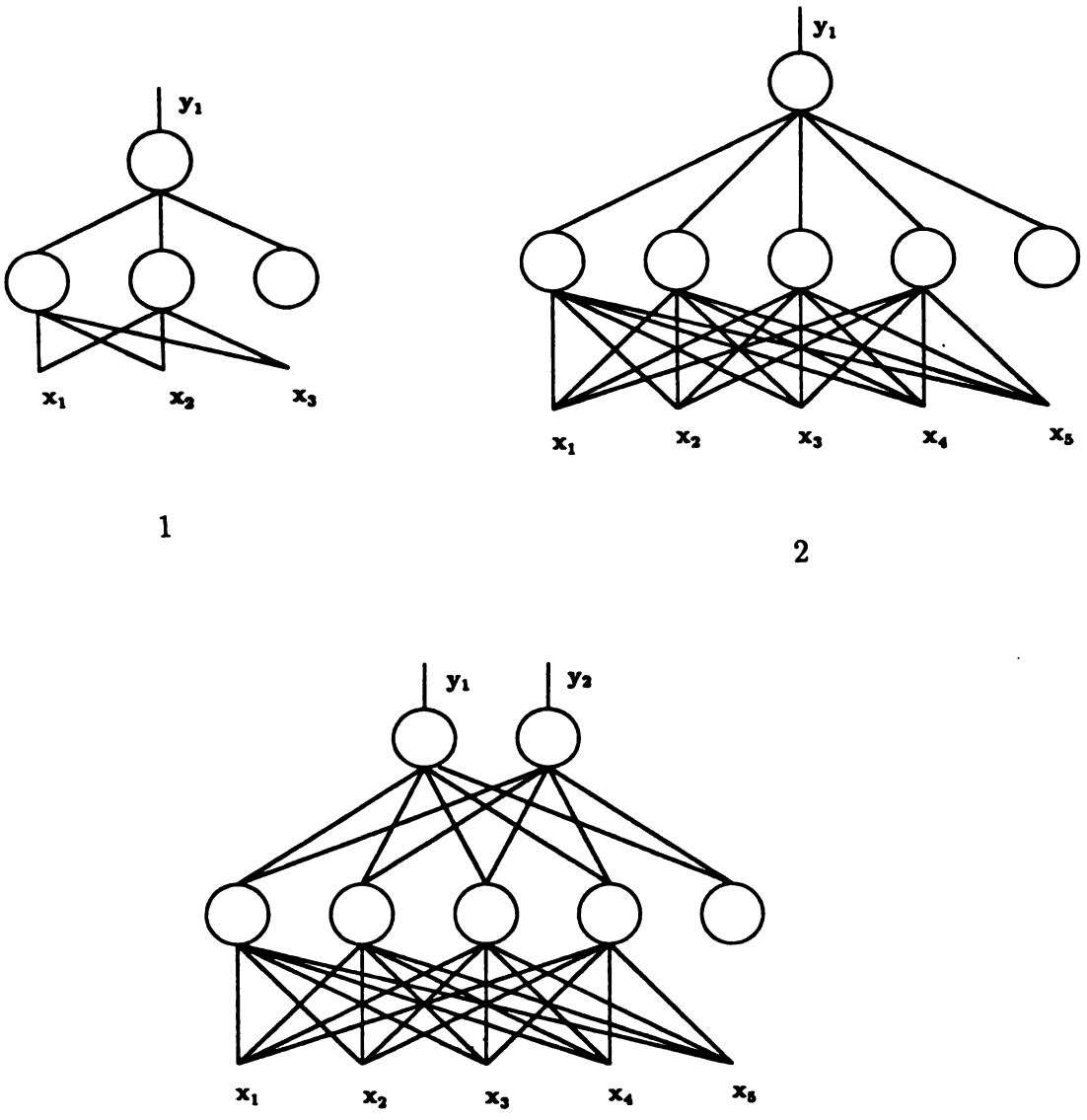

Figure 4.5: Network architectures used in the simulation studies. 1. the 2-bit parity checker, 2. the 4-bit parity checker, and 3. the 4-bit bit counter.

was repeated 100 times to select 100 sets of initial weights. The same 100 sets of weights were used for all 3 implementations. For the B-P algorithm, a factor of 0.04 was used in the weight updating equation. The A-S algorithm was implemented [6] using no weight change constraints. The forgetting factor for this and for the QR-WRLS implementation was 0.98. The QR-WRLS implementation used a weight constraint of 0.2. Thus the weight vector associated with each node was allowed to change at most by a magnitude of 0.2 in the Euclidean metric during each iteration.

The 4-bit parity checker network has four inputs, four hidden layer nodes and one output node. A bias input is also added to each layer. Two output functions were used. These were the same sigmoid function as above, and the logic activation function. The logic activation function is a three piece piecewise linear function,

$$f(x) = \begin{cases} 1 & if \quad x > 1 \qquad (4.1) \\ x & if \quad 0 \leq x \leq 1 \qquad (4.2) \\ 0 & if \quad x < 0 \end{cases}$$

The derivative of $S(\cdot)$ is trivially determined everywhere except at zero and one where it does not exist. This does not pose a problem in implementation if we let $\dot{S}(\alpha) = 1$ if $\alpha \in [0, 1]$ and zero else. Two sets of 100 random initial weights were used for the three implementations. The first set of weights was random as in the 2-bit parity checker, and the second set of weights was as described by Azimi-Sadjadi *et al.* in their paper. The A-S method selects the weights so that the outputs of the network will be between zero and one. This is done so that the derivative will not be zero and weight updating can take place. The 4-bit bit counter had four inputs, four hidden layer nodes and two outputs. An extra input was added to each layer. The logic activation function was used as the output function. Two sets of initial weights, random and those described by Azimi-Sadjadi, were used. The results are shown in

Table 4.2: Number of convergences per 100 sets of initial weights

Impementation	2in-1out	4in-1out				4in-2out	
	sigmoid	sigmoid		logic activation		logic activation	
	random weights	random weights	A-S weights	random weights	A-S weights	random weights	A-S weights
Q-R	78	5	5	51	57	1	16
Back-Prop	11	0	0	1	53	0	0
A-S	8	0	0	1	37	0	9

Table 4.2. The table shows the number of times each implementation found weights that solved the problem for the 100 initial weight sets.

Simulations were also run comparing the output error of each algorithm. In the resulting figures, the relative error in dB means the following: Let $\varepsilon(i)$ be the sum of the squared errors incurred in iteration i through the training patterns, averaged over the 100 initial weight sets. Then, plotted in the figures is $10\log(\varepsilon(i)/\mu)$ (dB), where μ is the maximum possible error at each iteration. The maximum possible error at one iteration is the sum of the square of the Euclidean distance between a training pattern and the output of the network associated with this training pattern.

Figure 4.6 shows the errors of the three X-OR implementations.

These results indicate a clear advantage for the QR-WRLS strategy. Algorithmic differences among the three implementations account for performance differences. One difference is the initialization needed for the MIL-WRLS equations. With the MIL-WRLS strategy, both the covariance matrix recursion and the weight vector recursion must be initialized using theoretically incorrect values. Because of initializations, the MIL-WRLS algorithm is not guaranteed to move the estimate in the direction of *greatest* decrease of E, or even of decreasing E, for the first few iterations. Of the two MIL-WRLS recursions, the weight recursion seems to be the most sensitive to the initialization problem. This is because $\boldsymbol{P}$ is initialized with large values,

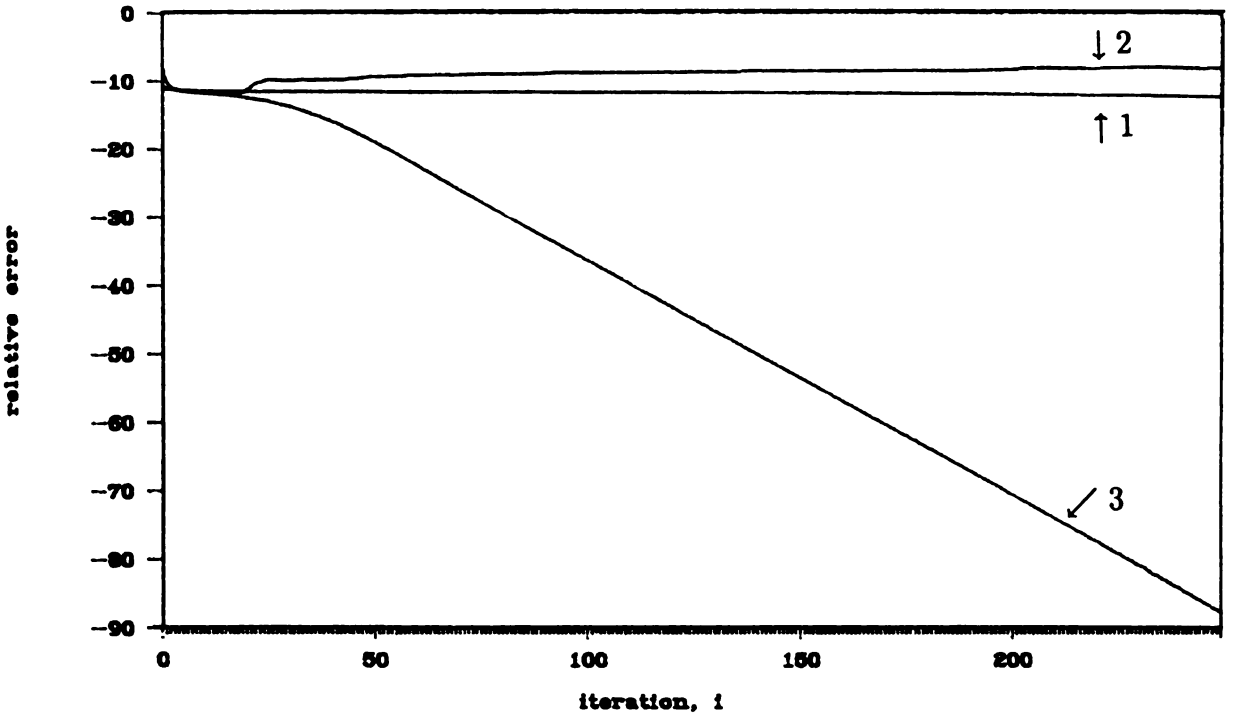

Figure 4.6: Average error in dB for the X-OR implementations vs. iteration number. 1. back-propagation; 2. A-S algorithm; 3. QR-WRLS.

$\boldsymbol{P}^{-1}$ is small and the effect of this initialization is relatively small. The weight recursion is sensitive to initialization because (2.50) depends explicitly upon $\boldsymbol{w}_k(i, n-1)$. The QR-WRLS algorithm has only an implicit dependence on the weights, as do all linearization algorithms, because the linearizations depend on the weights.

There is also a difference in the performance of the network using different functions for $S(\cdot)$. The logic activation function proved superior to the sigmoid in these experiments. This is probably because the error will always be positive using the sigmoid, but can be zero for the threshold logic activation function. No matter how the weights are adjusted, the output of the sigmoid will always be bounded by one, so that the training pattern outputs can never be matched exactly. With the threshold logic activation function, once the weights are adjusted so that the output is off the ramp (the linear region), the output can be zero or one in which case the difference between the training output and the actual output can be zero.

4.3 Layer Updating and Network Updating

This section compares BP, the node-wise updating algorithm, the layer-wise updating algorithm and the network-wise updating algorithm. It is divided into four subsections. The first shows the results of the four algorithms implementing a 2-bit parity checker and a 4-bit parity checker with different methods of selecting the initial weights. The second subsection shows the result of the three linearized algorithms with different forgetting factors. In the third subsection, results of simulations with many different, relatively small training sets are given for the four algorithms. The final subsection presents the results of the first three algorithms training with a large training set.

4.3.1 Initialization

One of the greatest problems in FNN training is the selection of the initial weights. As shown below, BP is very sensitive to the initial weights. Also, shown in the previous section is the fact that, other algorithms, such as A-S, suffer from the same problem. It is the purpose of this subsection to demonstrate that the layer-wise and network-wise weight updating algorithms are superior to BP in robustness to problems caused by "poor" initial weights. As will be seen in the simulations, both the layer-wise and network-wise algorithms can be initialized to very small values and perform very well. In fact the performance improves with very small initial values. This means that the initial weights can be initialized to small values, typically less than 0.01, without concern about the initial weights.

All four algorithms were used to train the 2-bit and 4-bit parity checker networks. The network architectures are as in the previous section. For the 2-bit parity checker, six different methods of choosing the initial weights were used. The first method was as in the previous section. Thus, 100 initial weight sets were chosen from the

uniform distribution $[-1, 1]$. Next, 100 initial weight sets were again chosen from the uniform distribution, but now over $[-.5, .5]$. This process was repeated, choosing 100 initial weight sets from uniform distributions over $[-.25, .25]$, $[-.1, .1]$, $[-.01, .01]$ and $[-.001, .001]$. Simulations were run for the four algorithms, and the results are shown in Table 4.3. The BP algorithm had a factor of 0.05 for the weight updating equations. The node-wise algorithm had a forgetting factor of 0.1 and a weight change constraint of 0.25. The layer-wise algorithm had a forgetting factor of 0.1 and a weight change constraint of 1.0. The network-wise algorithm had a forgetting factor of 0.3 and a weight change constraint of 1.0. The number of convergences out of a possible 100 is given. Also shown is the average number of iterations to convergence for the trials where the algorithm converged to a solution. Figures 4.7 through 4.10 show the average error of each of the algorithms for the different weight sets. Each of the six different weight sets is numbered, 1 is the weight set from the uniform distribution over $[-1, 1]$, 2 is the weight set over $[-.5, .5]$, 3 is the weight set over $[-.25, .25]$, 4 is the weight set over $[-.1, .1]$, 5 is the weight set over $[-.01, .01]$, and 6 is the weight set over $[-.001, .001]$. Each of the Figures 4.8 through 4.14, is numbered similarly. The simulations for the 4-bit parity checker are similar. 100 initial weights over the same distributions were chosen. The convergence results are shown in Table 4.4. The weight change constraints and forgetting factors were the same as the 2-bit case. Figures 4.11 through 4.14 show the average error of each of the algorithms for the varying weight sets.

The results show that the performance of BP and the node-wise algorithm suffer as the initial weights become small. The layer-wise and network-wise algorithms on the other hand show very good performance even with all initial weights smaller than 0.001 in magnitude. Because of the nature of the problem, gradient following algorithms are sensitive to the initial weights. Some initial weights will lead to a solution, and others will not. What we would like is one set of weights that would

Table 4.3: Convergence results for various initial weights, 2-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence.

Weight variation	BP		Node-wise		Layer-wise		Network-wise	
	A	B	A	B	A	B	A	B
[-1,1]	1	-	70	23.59	97	14.84	86	96.78
[-.5,.5]	0	-	93	33.85	99	18.74	93	122.18
[-.25,.25]	0	-	99	72.95	99	11.55	94	150.37
[-.1,.1]	0	-	80	156.85	99	11.22	94	166.61
[-.01,.01]	0	-	7	393.57	97	12.08	98	81.97
[-.001,.001]	0	-	5	376.6	96	17.45	93	32.51

Table 4.4: Convergence results for various initial weights, 4-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence.

Weight variation	BP		Node-wise		Layer-wise		Network-wise	
	A	B	A	B	A	B	A	B
[-1,1]	0	-	38	207.90	59	65.20	35	238.17
[-.5,.5]	0	-	9	301.67	76	65.16	38	197.03
[-.25,.25]	0	-	0	-	76	52.01	42	148.95
[-.1,.1]	0	-	0	-	82	47.43	45	138.62
[-.01,.01]	0	-	0	-	89	26.01	50	100.52
[-.001,.001]	0	-	0	-	87	29.92	66	121.89

lead to a solution in all cases. This is, of course, impossible, but the large number of convergences of the layer-wise and network-wise algorithm for very small initial weights indicate that very small weights can be used in practice to approximate the "ideal" initial weights.

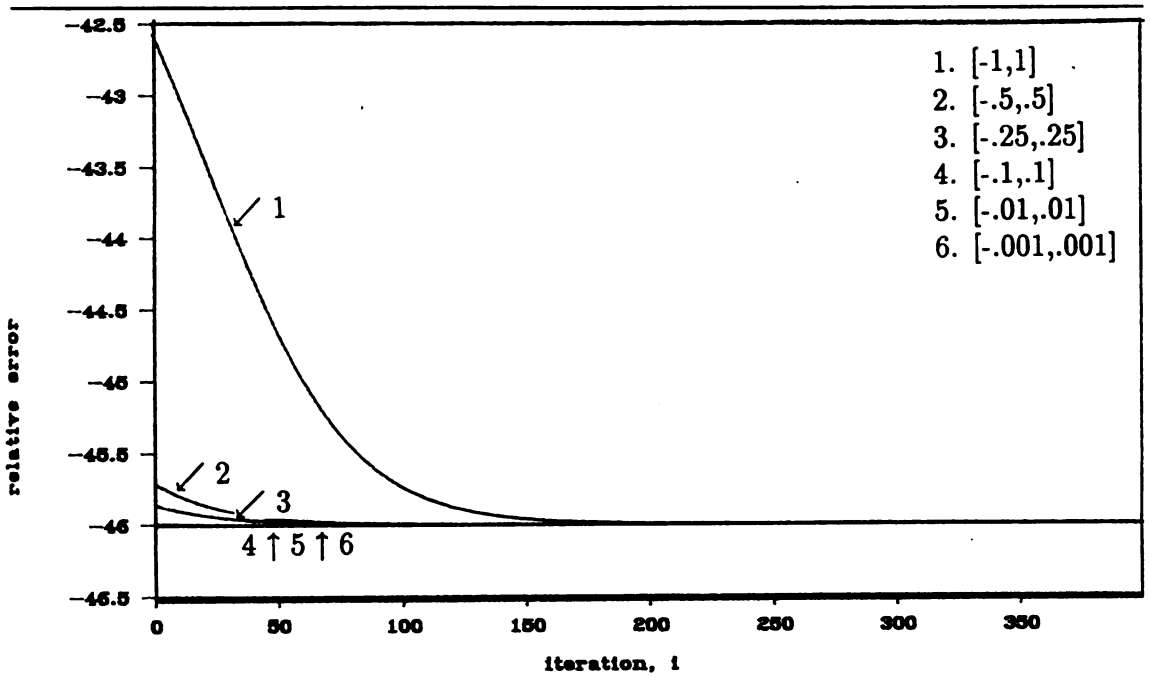

Figure 4.7: Average error in dB for the BP implementation of the 2-bit parity checker vs. iteration number using different methods of selecting initial weights.

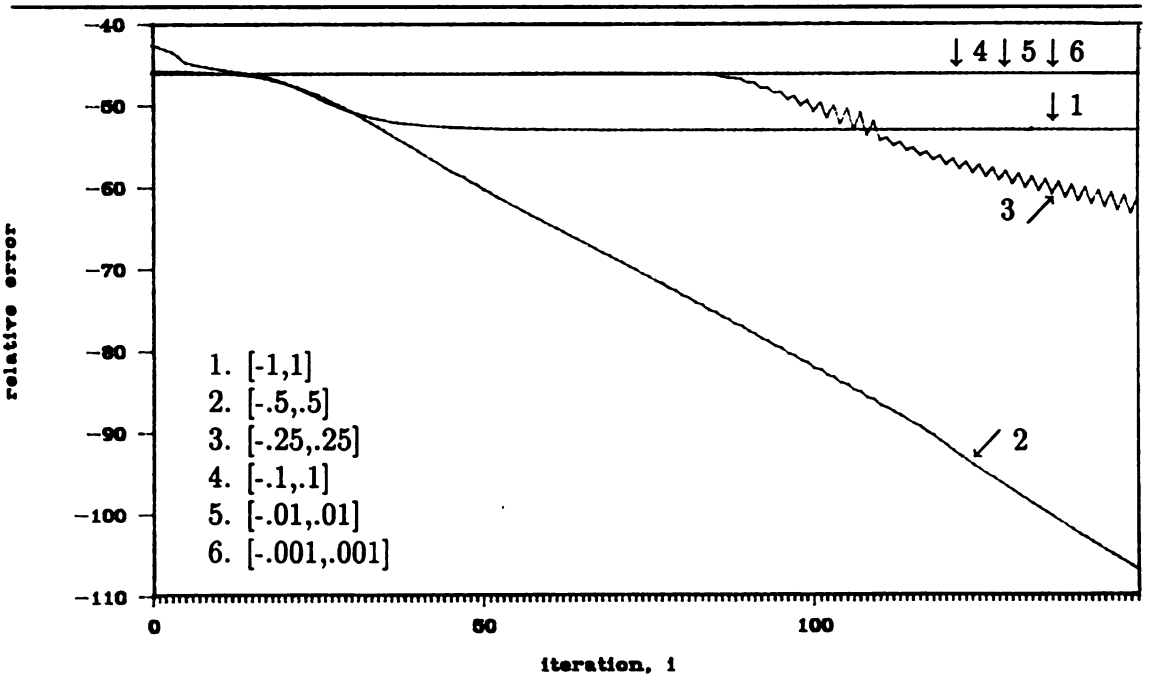

Figure 4.8: Average error in dB for the node-wise implementation of the 2-bit parity checker vs. iteration number, using different methods of selecting initial weights.

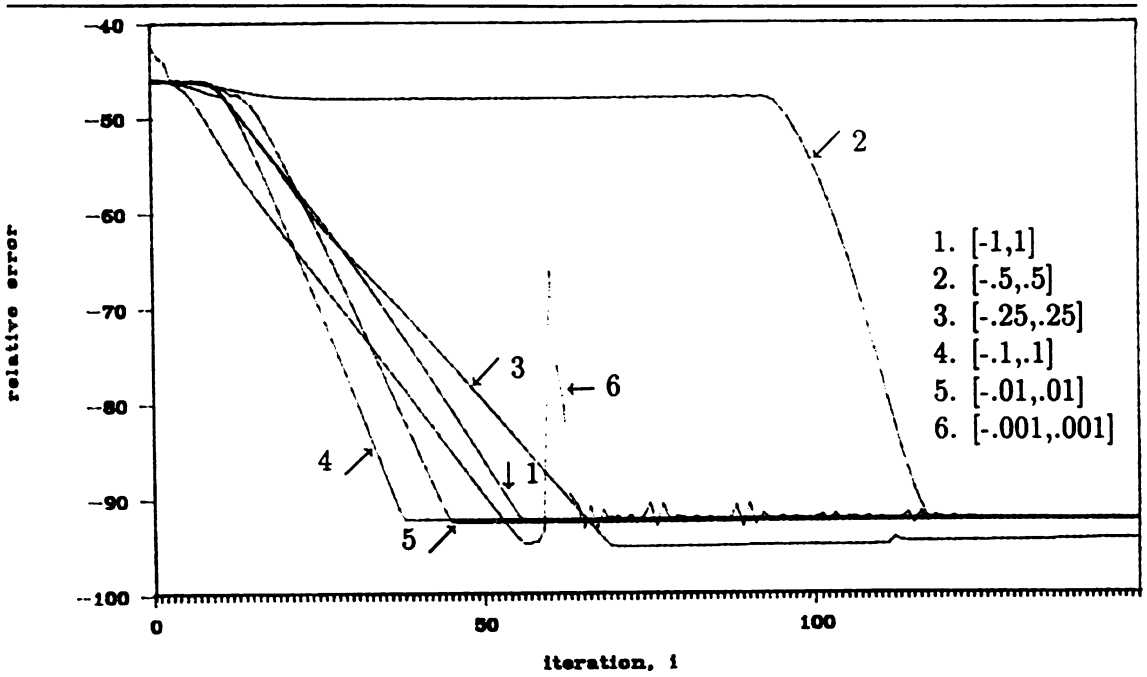

Figure 4.9: Average error in dB for the layer-wise implementation of the 2-bit parity checker vs. iteration number, using different methods of selecting initial weights.

4.3.2 Forgetting Factor

This subsection considers the performance of the node-wise, layer-wise and network-wise algorithms with different forgetting factors. When training a FNN with QR-WRLS, some method of "forgetting" past linearized training patterns must be used in the equations. This forgetting is necessary because old linearized training patterns are not useful in determining the gradient of the error surface at the present weights. As mentioned previously, implementation with QR-WRLS allows a variety of forgetting methods. The one that has proved to be very good is a forgetting factor of zero. In other words, with all previous training patterns completely "forgotten" (removed from the estimation process). The forgetting factor is often problem dependent, and one that works with one training set may not work well with another training set. Using a very small or a zero forgetting factor means that one variable has been eliminated, simplifying training.

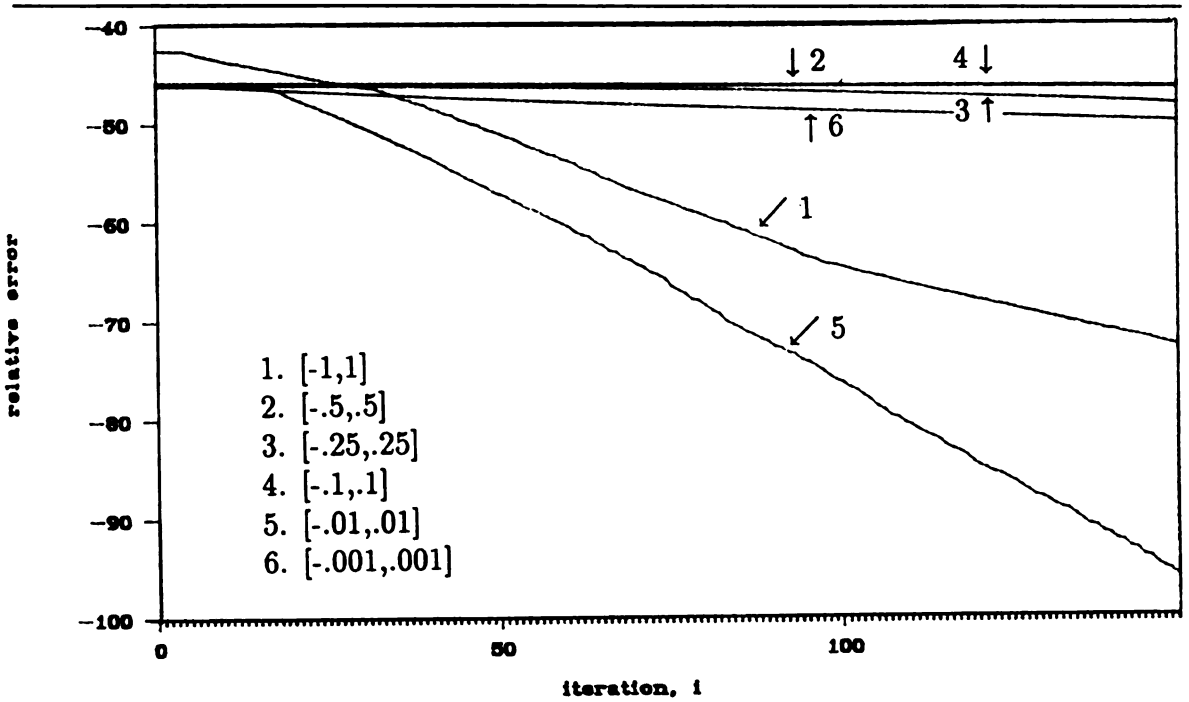

Figure 4.10: Average error in dB for the network-wise implementation of the 2-bit parity checker vs. iteration number, using different methods of selecting initial weights.

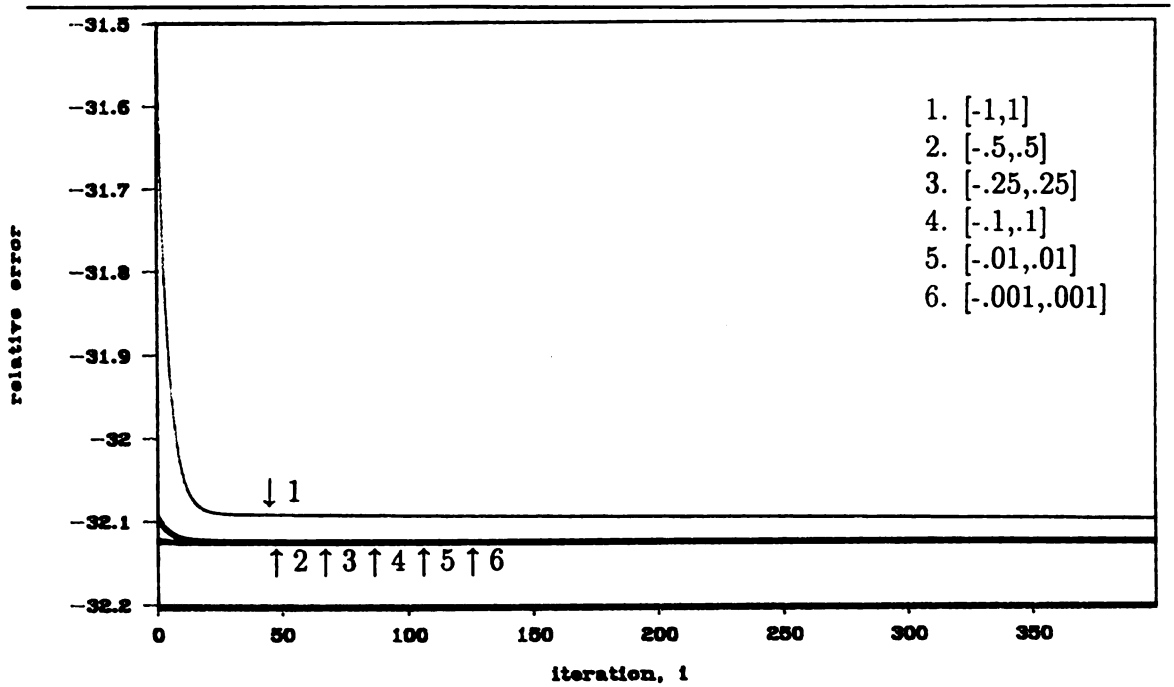

Figure 4.11: Average error in dB for the BP implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights.

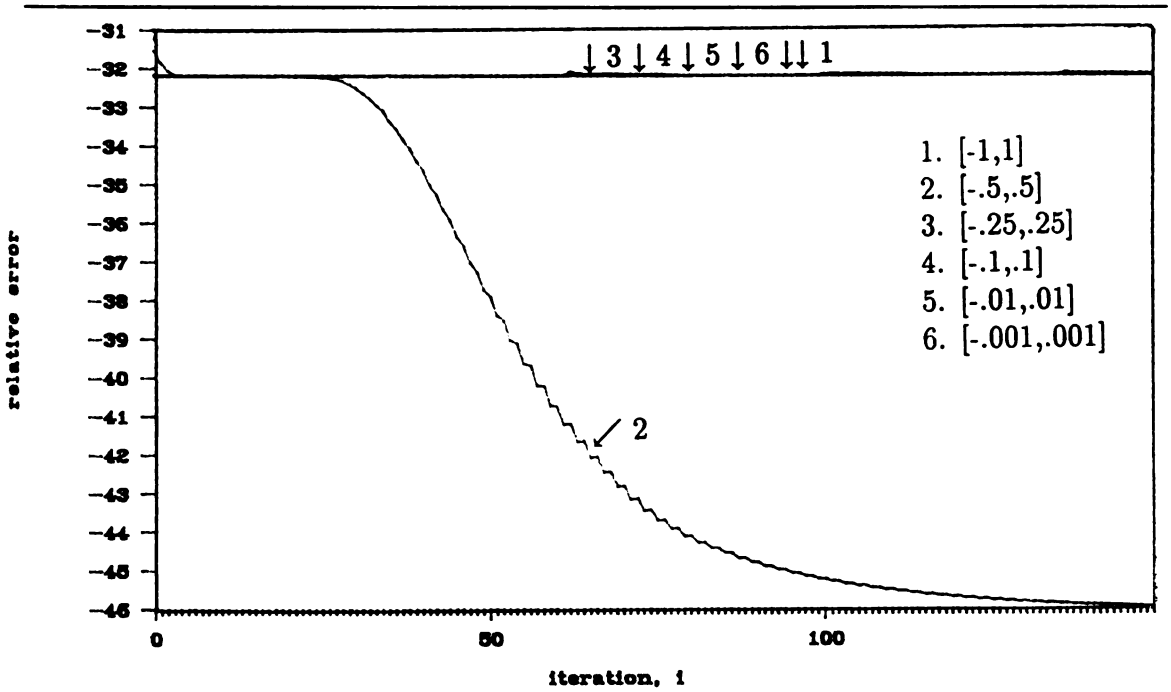

Figure 4.12: Average error in dB for the node-wise implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights.

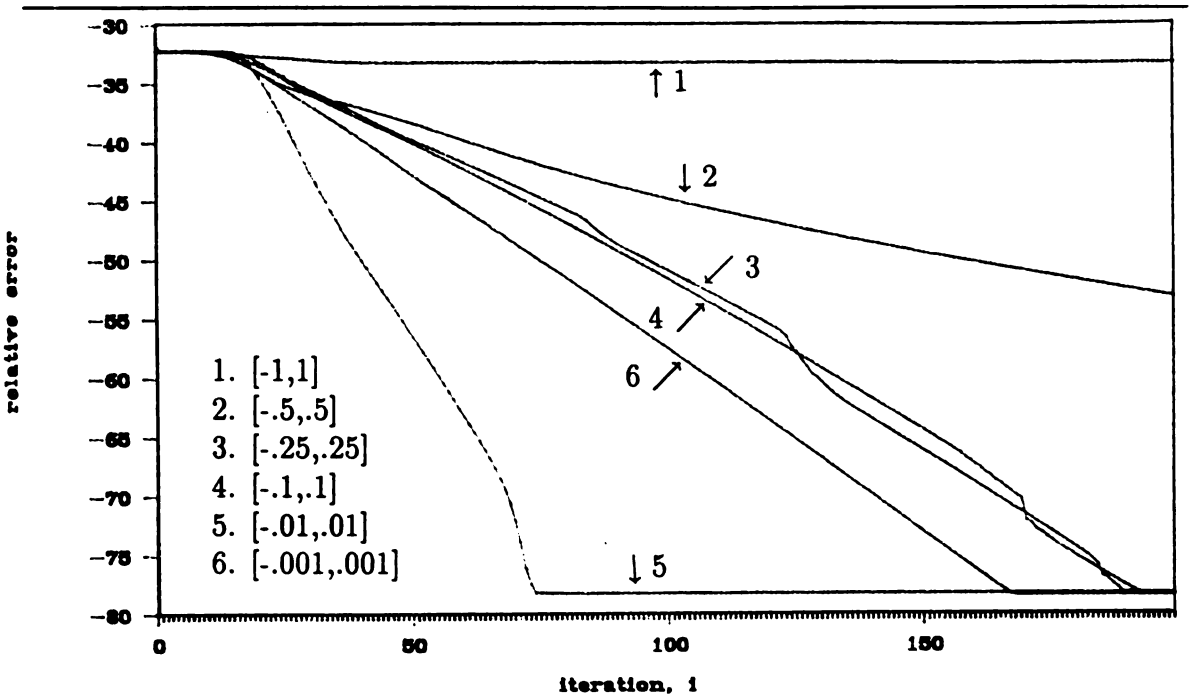

Figure 4.13: Average error in dB for the layer-wise implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights.

The forgetting factor and weight change constraint that give good performance are related. This means that if the forgetting factor is optimized for a certain training set and weight change constraint, then changing the weight change constraint will mean that the optimum forgetting factor will probably also change. In the simulations it is noted that for good performance, a small weight change constraint must be used with a small forgetting factor. This relation is shown in Figure 4.15. Here the errors of the node-wise algorithm implementing a 2-bit parity checker using different forgetting factors and different weight constraints. The number of convergences out of a possible 100 for each of the settings was 78 for simulation 1, 60 for simulation 2, 56 for simulation 3 and 64 for simulation 4. It is apparent that the parameters which yield the most convergences do not necessarily lead to the lowest average error. This trade-off must be considered when implementing the algorithms.

Even though there is a relation between the forgetting factor and the weight

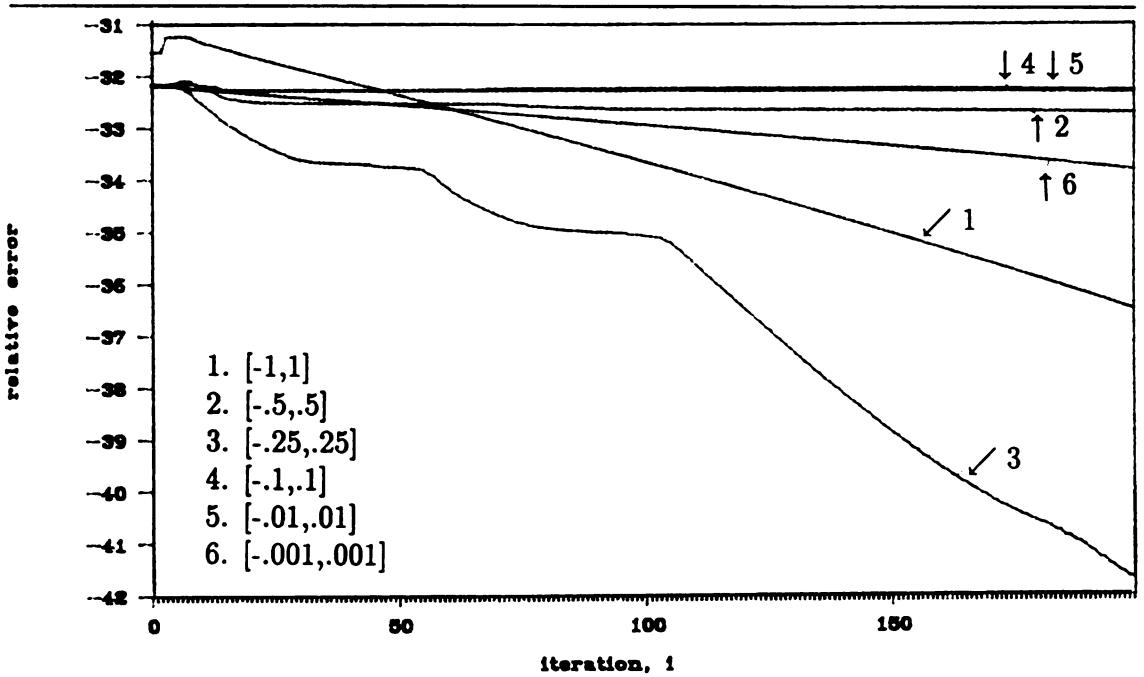

Figure 4.14: Average error in dB for the network-wise implementation of the 4-bit parity checker vs. iteration number, using different methods of selecting initial weights.

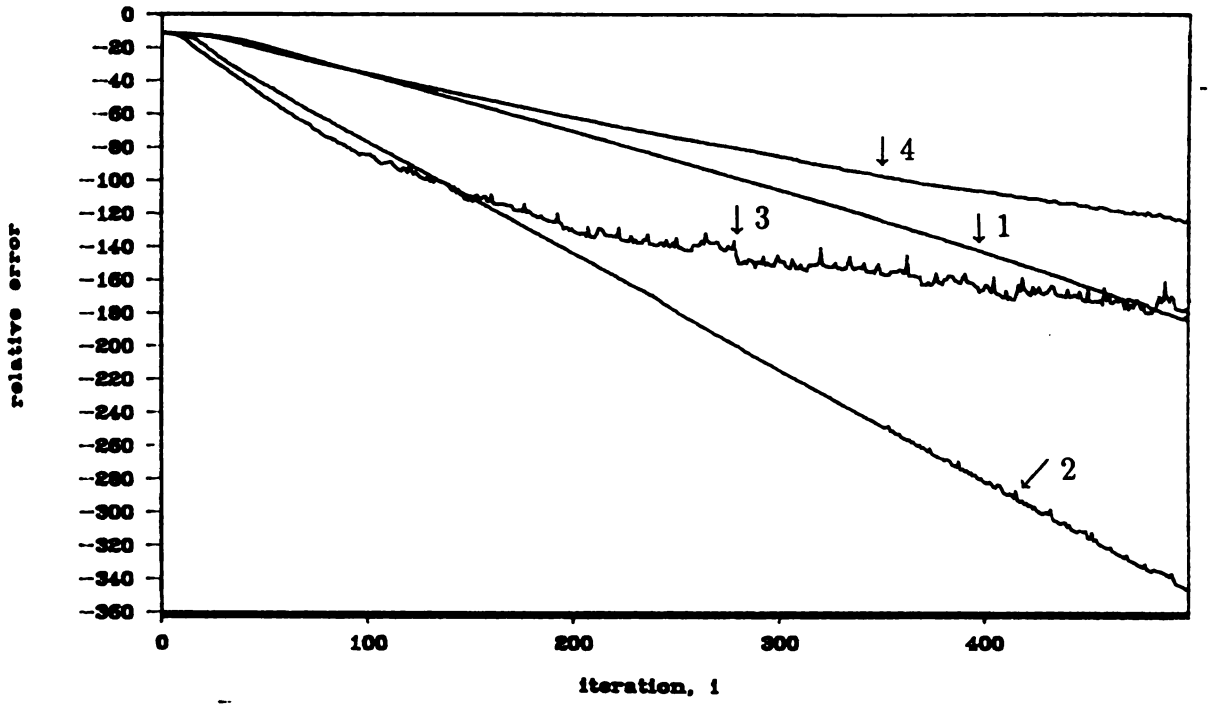

Figure 4.15: Average error in dB for the QR-WRLS X-OR implementation vs. iteration number, using different forgetting factors and weight change constraints. 1. $\nu = 0.98, \gamma = 0.2$; 2. $\nu = 0.98, \gamma = 1.0$; 3. $\nu = 0.1, \gamma = 1.0$; 4. $\nu = 0.1, \gamma = 0.2$; where ν is the forgetting factor and γ is the weight constraint.

constraint, for simplicity in these simulations, the weight constraint was held constant as the forgetting factor was varied. As in the previous subsection, a 2-bit parity checker and a 4-bit parity checker are used in the simulations. 100 sets of initial weights were used. The distribution for the weight selection for the 2-bit network was $[-.25, .25]$ for the node-wise algorithm, $[-.1, .1]$ for the layer-wise algorithm, and $[-.01, .01]$ for the network-wise algorithm. These initial weight distributions were selected using the results of the previous section, to obtain the best results for each algorithm. As in the previous section, the weight change constraint was 0.25 for the node-wise algorithm, and 1.0 for the layer-wise and network-wise algorithms. For the 4-bit network the distribution for the weight selection was was $[-1.0, 1.0]$ for the node-wise algorithm, and $[-.01, .01]$ for the layer-wise and network-wise algorithms. Again, these initial weight distributions were selected to optimize the results for each algorithm. The weight change constraint was the same as the 2-bit network. The results for different forgetting factors are shown in Table 4.5 for the 2-bit network and Table 4.6 for the 4-bit network. Figures 4.16 through 4.18 show the average error of each algorithm for the different forgetting factors for the 2-bit network. Figures 4.19 through 4.21 show the average error of each algorithm for the different forgetting factors for the 4-bit network. In each of these figures the the different forgetting factors are numbered. Trial runs with forgetting factor 0.75 are numbered 1, those with forgetting factor 0.5 are numbered 2, those with forgetting factor 0.25 are numbered 3 and those with forgetting factor 0.1 are numbered 4. To show the interdependence of the weight change constraint and the forgetting factor, the simulations using a 4- bit parity checker were run again using different weight constraints. The new weight constraints were 0.1 for the node-wise algorithm, and 0.4 for the layer-wise and network-wise algorithms. Results are shown in Table 4.7. Figures 4.22 through 4.24 again show the average errors for different forgetting factors. The same numbering as above for the different forgetting factors applies.

For the simple 2-bit network, the performance of the training algorithms does not exhibit much change across different forgetting factors. For the 4-bit network, there is more variance. Table 4.6 shows that a large forgetting factor is reduces the iterations to convergence. When the weight change constraint is decreased, the performance of the layer-wise and network-wise algorithm improves, and smaller forgetting factors improves performance. The performance of the node-wise algorithm with the smaller weight change is surprising. There are no convergences for any of the forgetting factors tried. Although the results indicate that the layer and network-wise algorithms have very good performance using small forgetting factors and weight change constraints, the most important result may be the fact that the performance of these two algorithms, especially the layer-wise algorithms, is quite impervious to changes in forgetting factors and weight change constraints. This, along with the small initial weights, greatly simplifies implementation.

The simulations presented in the last two sections deal with the problems of weight initialization and the choosing of a forgetting factor. 2-bit and 4-bit parity checkers are used for training. The initial weights and the forgetting factor were varied to show that the layer-wise algorithm has good performance with small initial weights and small forgetting factors. The BP and node-wise algorithm were shown to be very sensitive to these variables and performance varied greatly as they changed.

4.3.3 Small Training Sets

In this section many small training sets are used to compare the BP, the node-wise, the layer-wise, and the network-wise training algorithms. These small training sets had with two inputs, and from twenty to thirty training patterns each. For each training set, the number of iterations required to correctly classify all the training patterns was determined. Simulations were run using two layer networks for all four algorithms. The number of hidden nodes was varied from two to four for the simple

Table 4.5: Convergence results for various forgetting factors, 2-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence.

Forgetting factor	Node-wise		Layer-wise		Network-wise	
	A	B	A	B	A	B
0.75	97	110.87	96	7.5	84	37.89
0.5	97	75.98	100	12.28	86	44.51
0.25	96	69.33	100	9.59	91	52.55
0.1	96	67.04	98	9.75	84	86.63

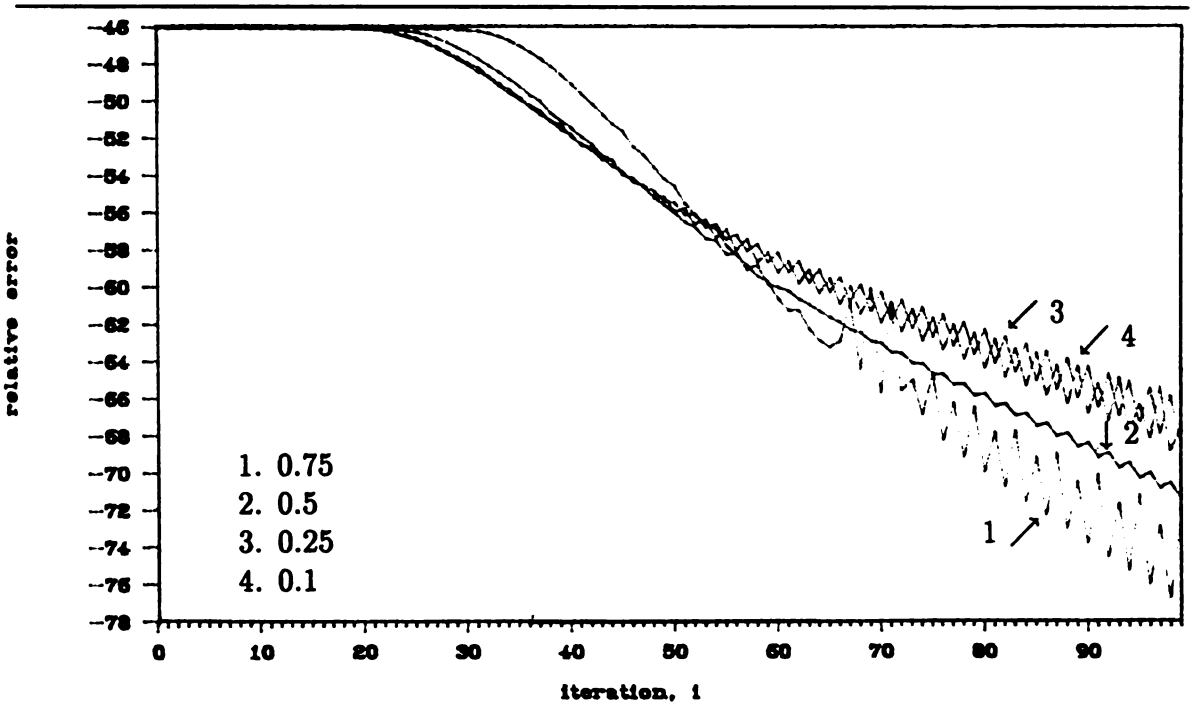

Figure 4.16: Average error in dB for the node-wise implementation of the 2-bit parity checker vs. iteration number, using different forgetting factors.

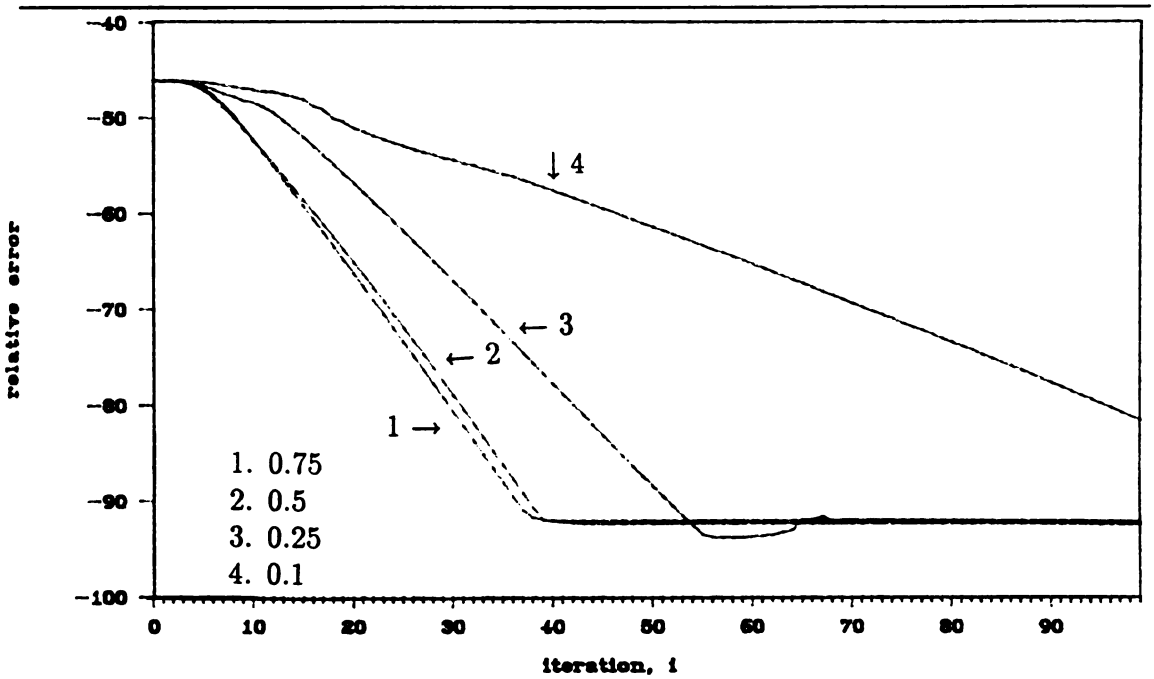

Figure 4.17: Average error in dB for the layer-wise implementation of the 2-bit parity checker vs. iteration number, using different forgetting factors.

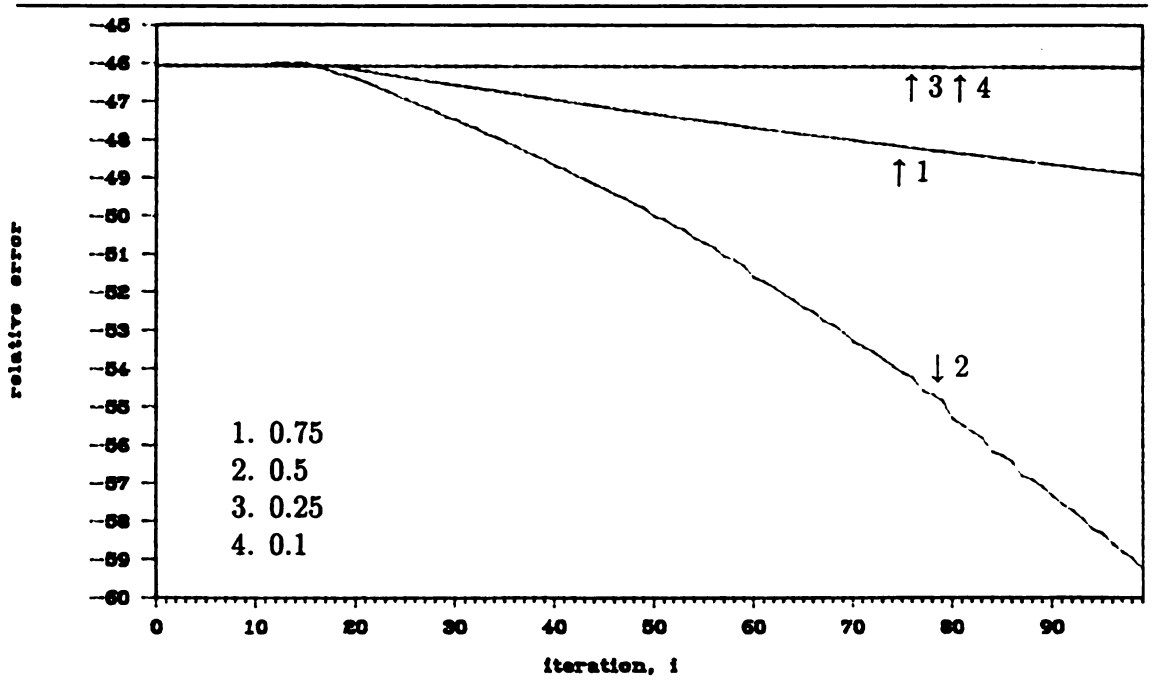

Figure 4.18: Average error in dB for the network-wise implementation of the 2-bit parity checker vs. iteration number, using different forgetting factors.

Table 4.6: Convergence results for various forgetting factors, 4-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence.

Forgetting factor	Node-wise		Layer-wise		Network-wise	
	A	B	A	B	A	B
0.75	32	202.94	95	18.6	69	100.75
0.5	40	224.95	91	28.03	73	119.32
0.25	40	234.73	93	20.17	50	118.72
0.1	42	263.98	90	27.26	50	151.1

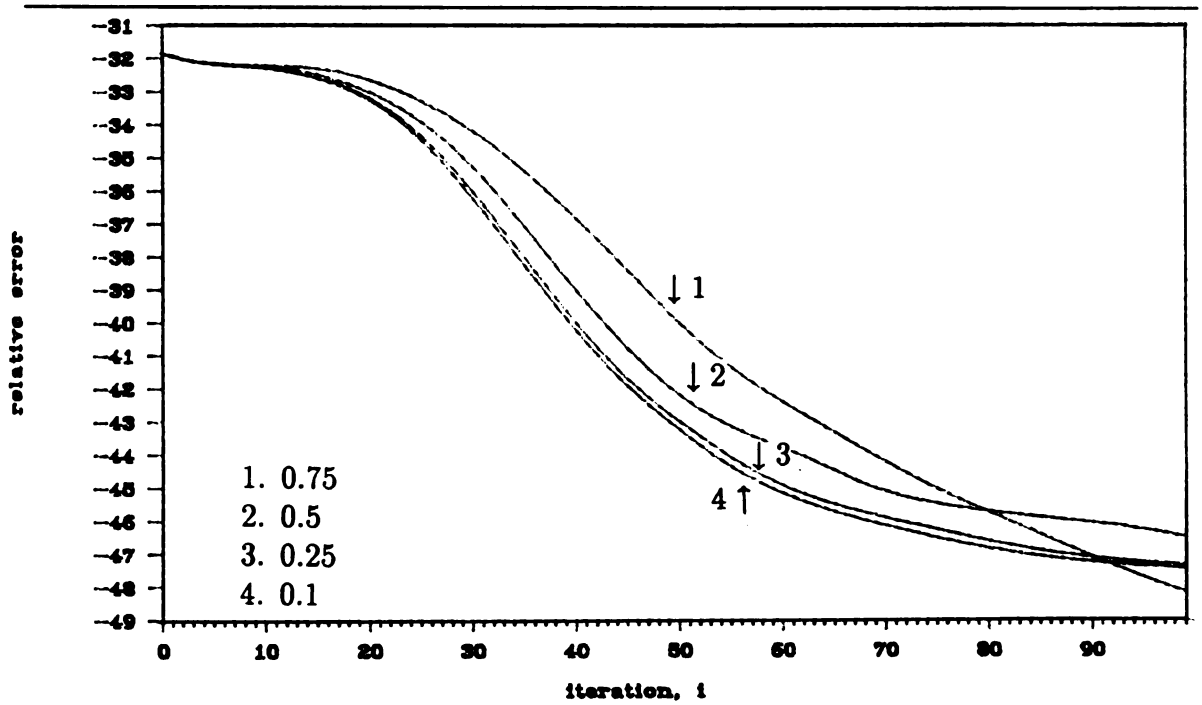

Figure 4.19: Average error in dB for the node-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors.

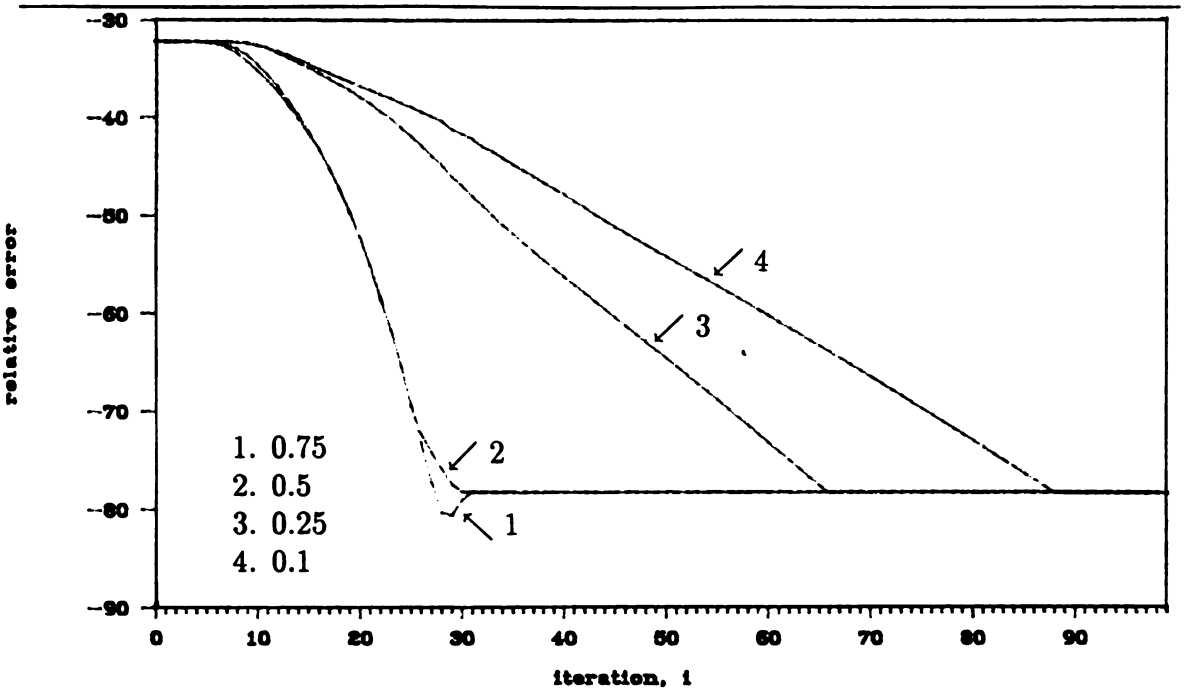

Figure 4.20: Average error in dB for the layer-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors.

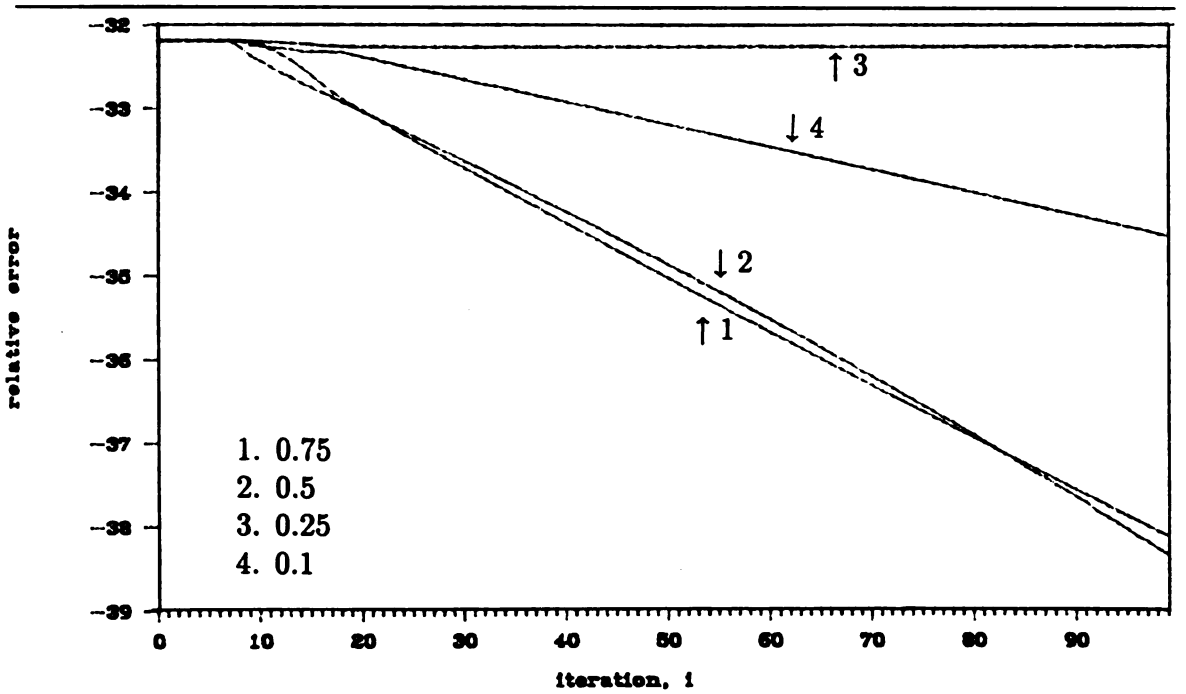

Figure 4.21: Average error in dB for the network-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors.

Table 4.7: Convergence results for various forgetting factors, 4-bit parity checker. A is the number of convergences out of 100, and B is the average number of iterations to convergence.

Forgetting factor	Node-wise		Layer-wise		Network-wise	
	A	B	A	B	A	B
0.75	0	-	91	28.89	67	106.55
0.5	0	-	95	24.42	63	93.06
0.25	0	-	97	31.52	57	132.02
0.1	0	-	88	26.51	97	160.04

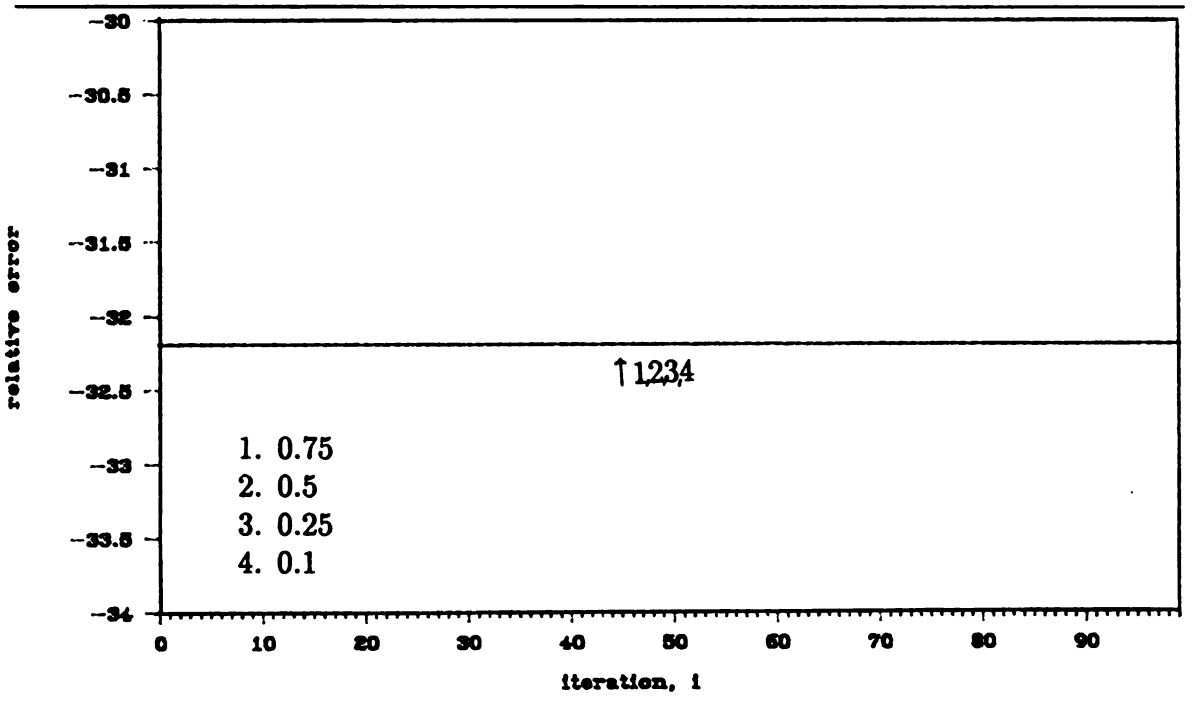

Figure 4.22: Average error in dB for the node-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors.

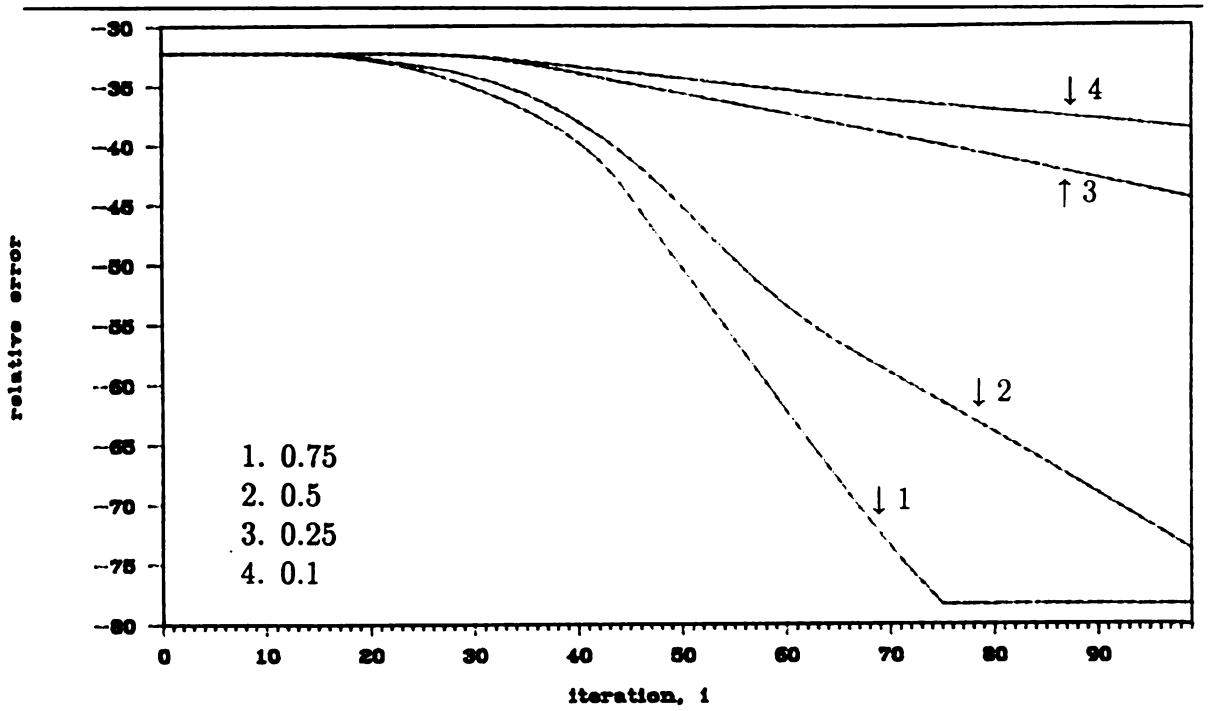

Figure 4.23: Average error in dB for the layer-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors.

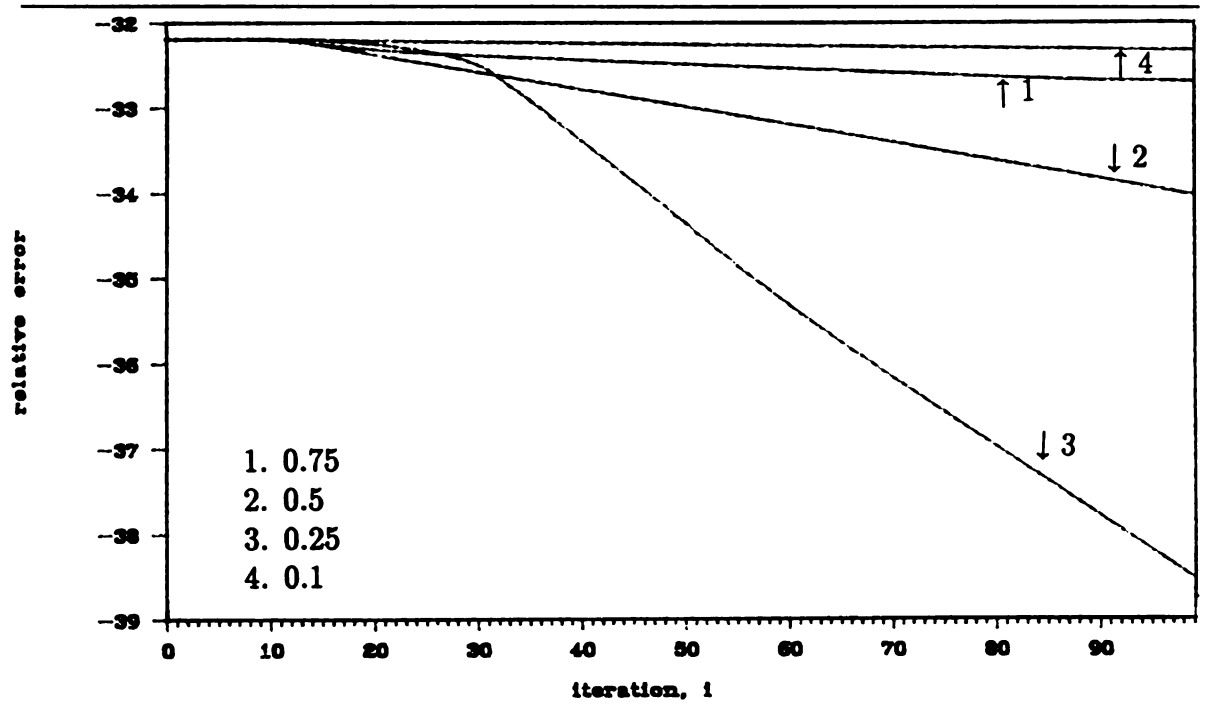

Figure 4.24: Average error in dB for the network-wise implementation of the 4-bit parity checker vs. iteration number, using different forgetting factors.

training sets. Up to seven hidden nodes were used for the more complicated training sets. Since the network-wise algorithm can have only one output, only simulations for training sets with two classes could be used. A particular combination of training pattern set and number of hidden layer nodes that was not run are left blank. A dash means that the algorithm failed to find weights that correctly classified all the training patterns. The results are shown in Tables 4.7 through 4.10.

Training set A, shown if Figure 4.25, is a simple linearly separable set using training patterns from two classes. The two classes are separated by one of the axes. Training set B is the same as A, only the classes to which each training pattern belongs has been interchanged. This was done for many of the training sets. Thus, training sets C and D; E and F; G and H; and I and J are similar, only interchanging the class of the training patterns. This was done to determine if the training algorithms favored positive or negative weights. If an algorithm found weights so that the network correctly classifies training set A, then changing the sign on the weights means that the network can now correctly classify training set B. Similarly for training sets C and D; E and F; G and H; and I and J. Training set C is linearly separable, but not by any of the axes as in training set A. Training set E is linearly separable, however the separating line cannot run trough the origin as is the case for training sets A through D. Training set G is the first training set that is not linearly separable. Training set L has training pairs divided into four classes. The training pairs are again linearly separable and BP and the node-wise algorithms again perform very well. Training set N has ten classes of training pairs. The training pairs were taken from the large training set used in the next section.

The first result to note is the poor performance of the network-wise training algorithm. It seems that the algorithm is interesting from a theoretical point of view, but is not practical for use in training without further work. From the tables, it can be seen that the node-wise and BP performed very well for the linearly separable training

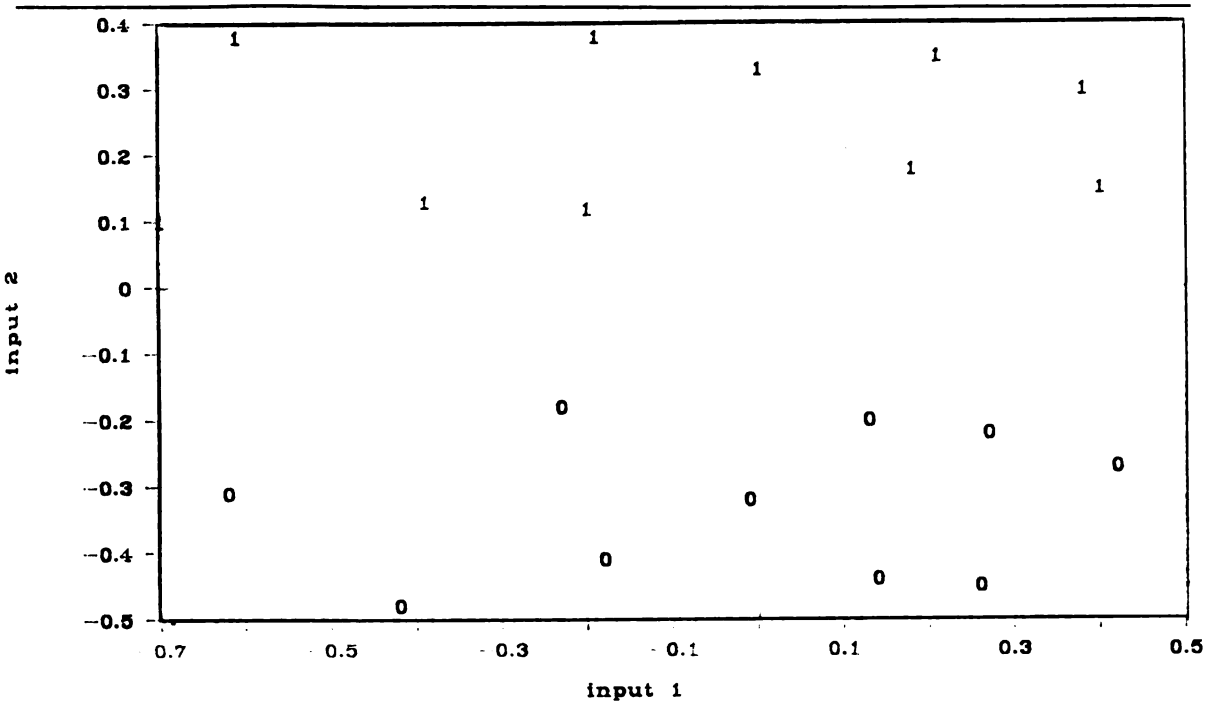

Figure 4.25: Training set A.

sets. When the training sets were not linearly separable, however the advantages of the layer-wise algorithm are apparent. Most interesting is the fact that the number of iterations to convergence for the layer-wise algorithm for training sets A through K vary only slightly. Also, as shown in the tables, the layer-wise algorithm is the only one to find weights that correctly classifiy all training patterns for training set N. This provides evidence to what has been mentioned previously, that changing all the weight of the same layer has a great advantage over changing the weights independently.

4.3.4 Large Training Set

In this section a large training set was used to compare the BP, the node-wise and the layer-wise training algorithms. The training set has 519 training pairs. As can be seen in Figure 4.34, the training pairs are overlapping and so the correct classification of all training pairs is not possible. The layer-wise algorithm performed the best. It has the

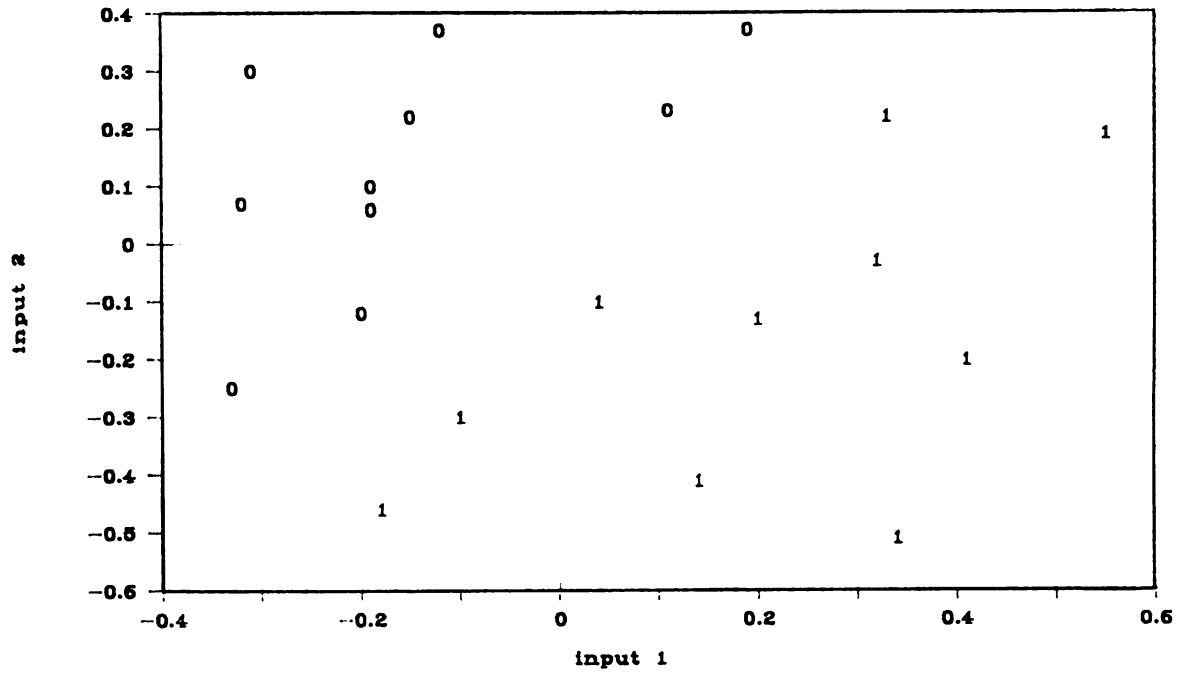

Figure 4.26: Training set C.

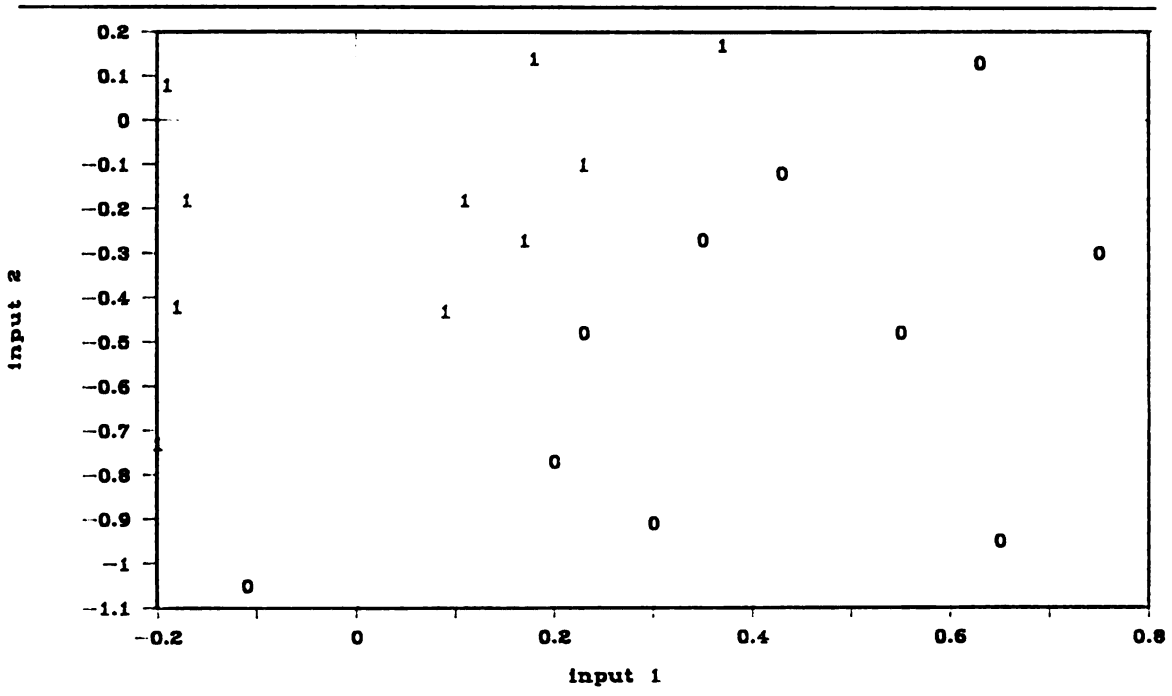

Figure 4.27: Training set E.

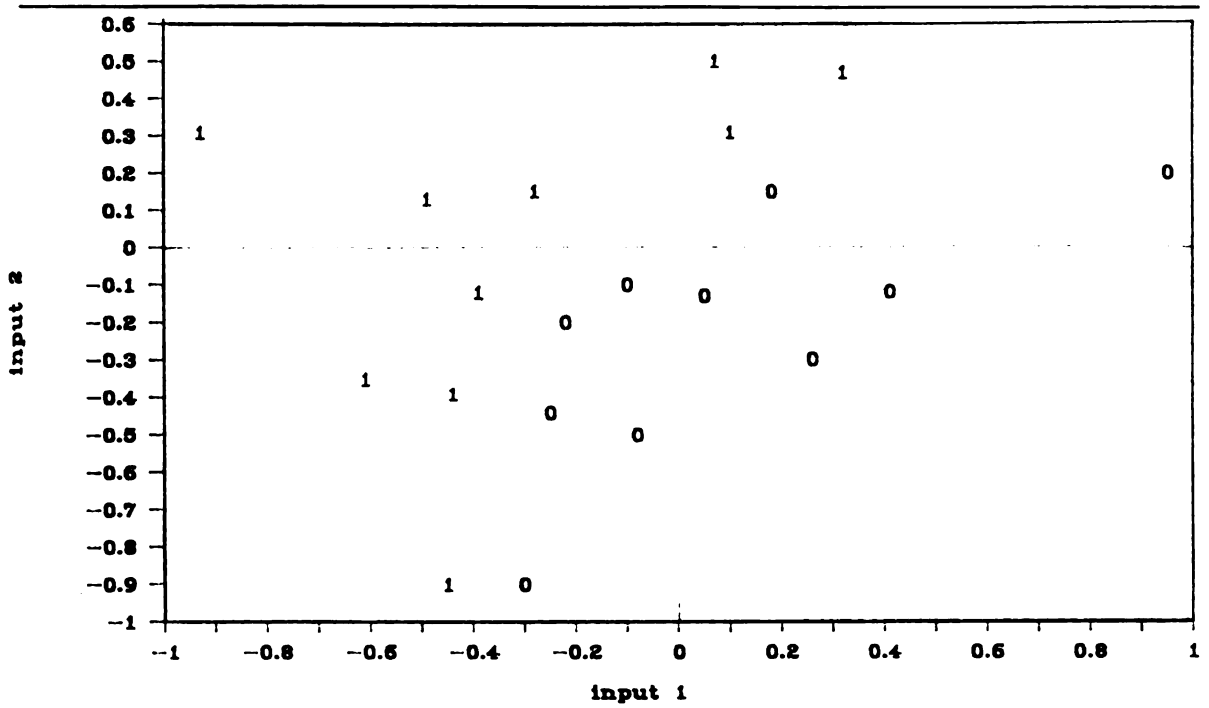

Figure 4.28: Training set G.

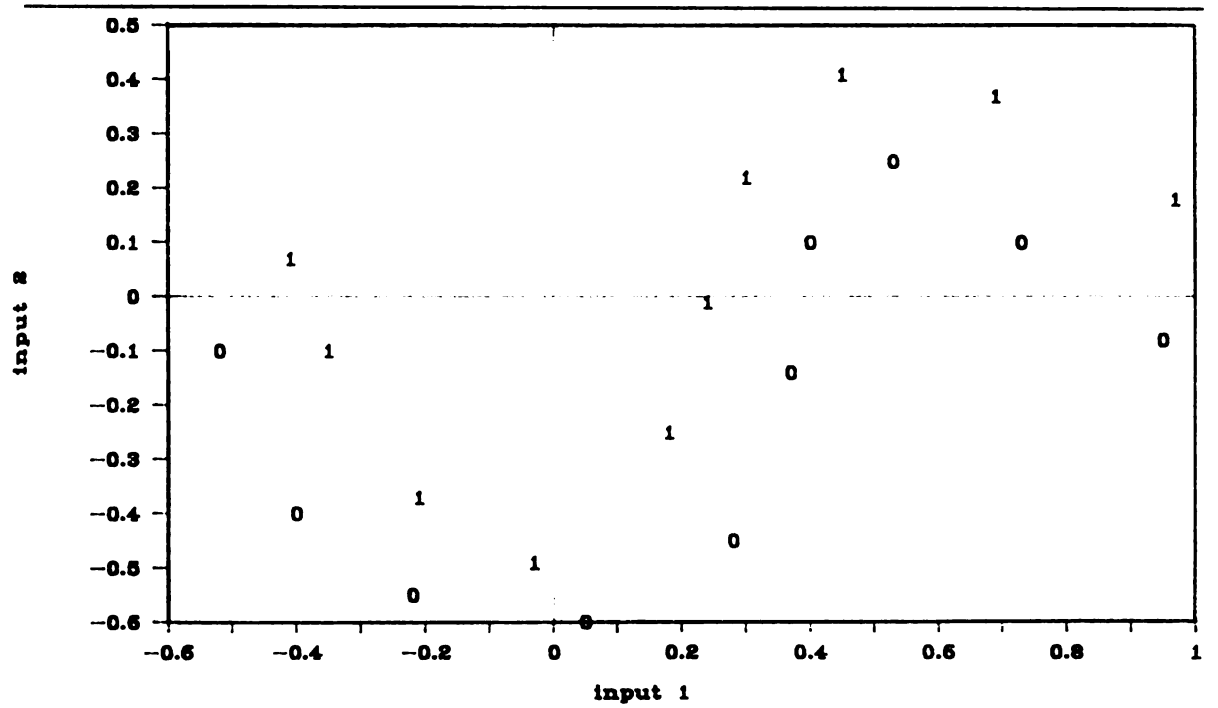

Figure 4.29: Training set I.

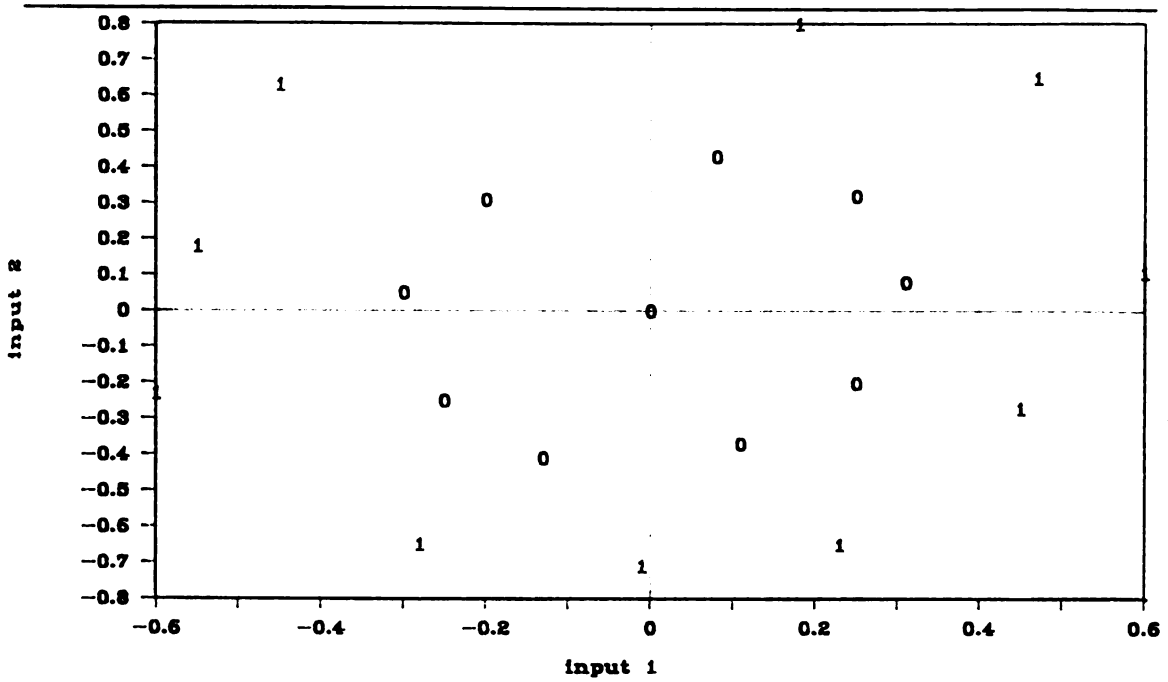

Figure 4.30: Training set K.

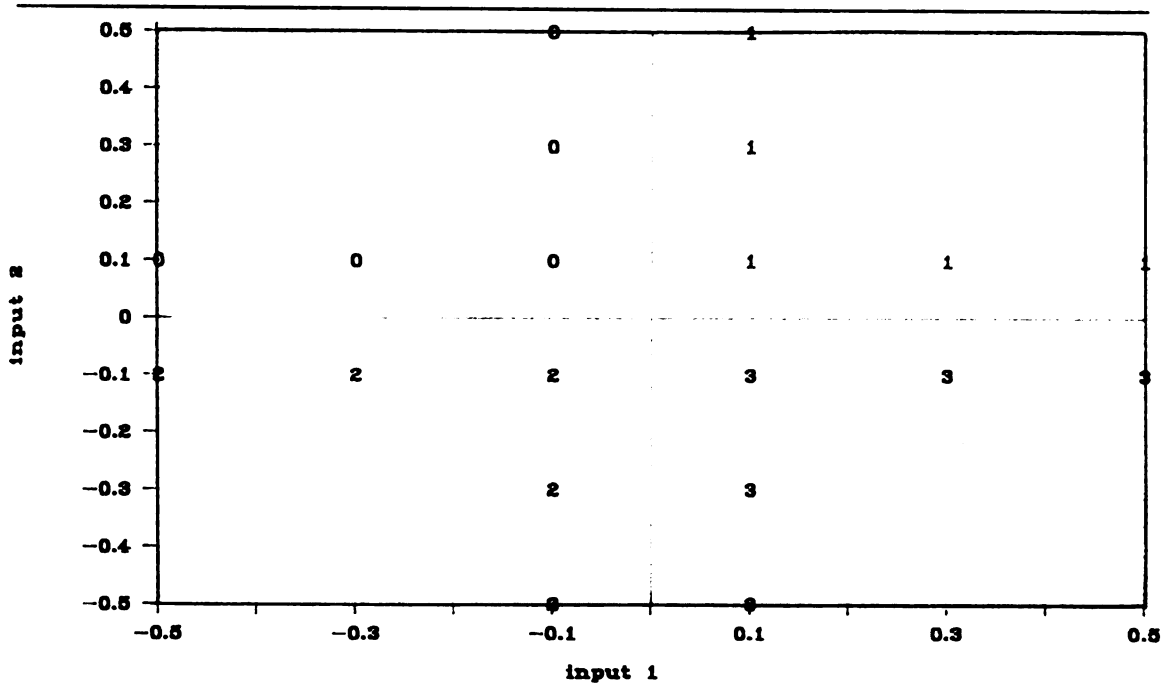

Figure 4.31: Training set L.

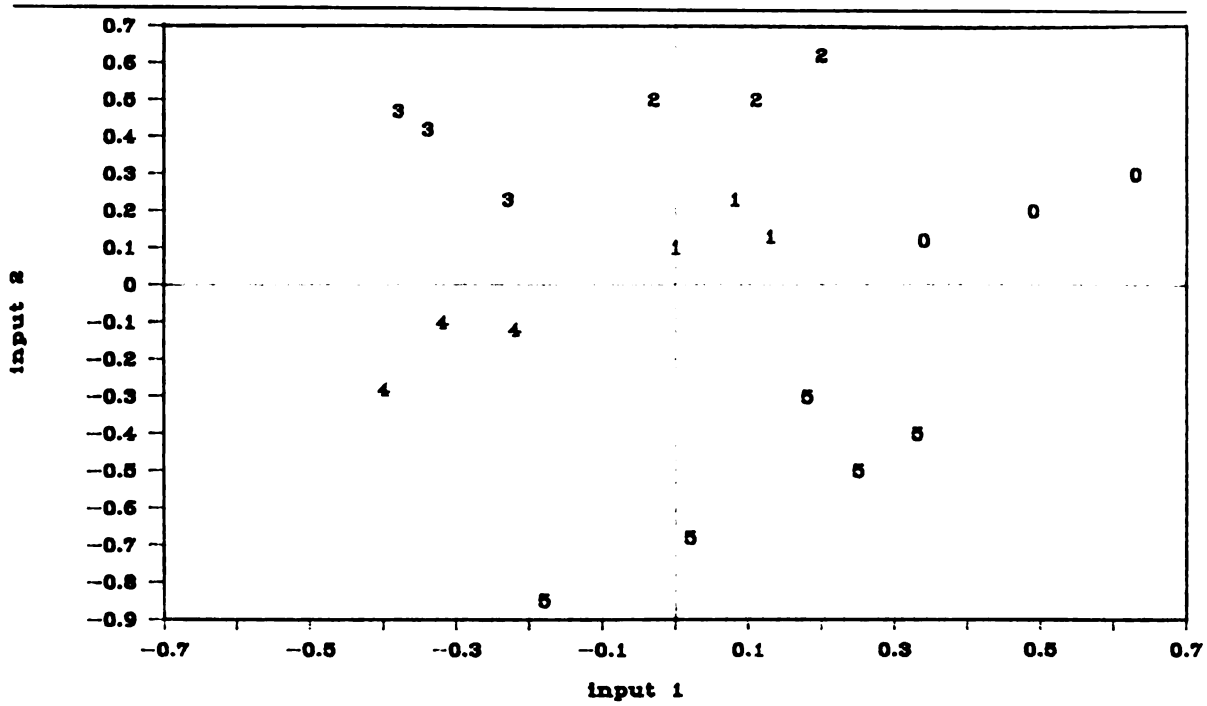

Figure 4.32: Training set M.

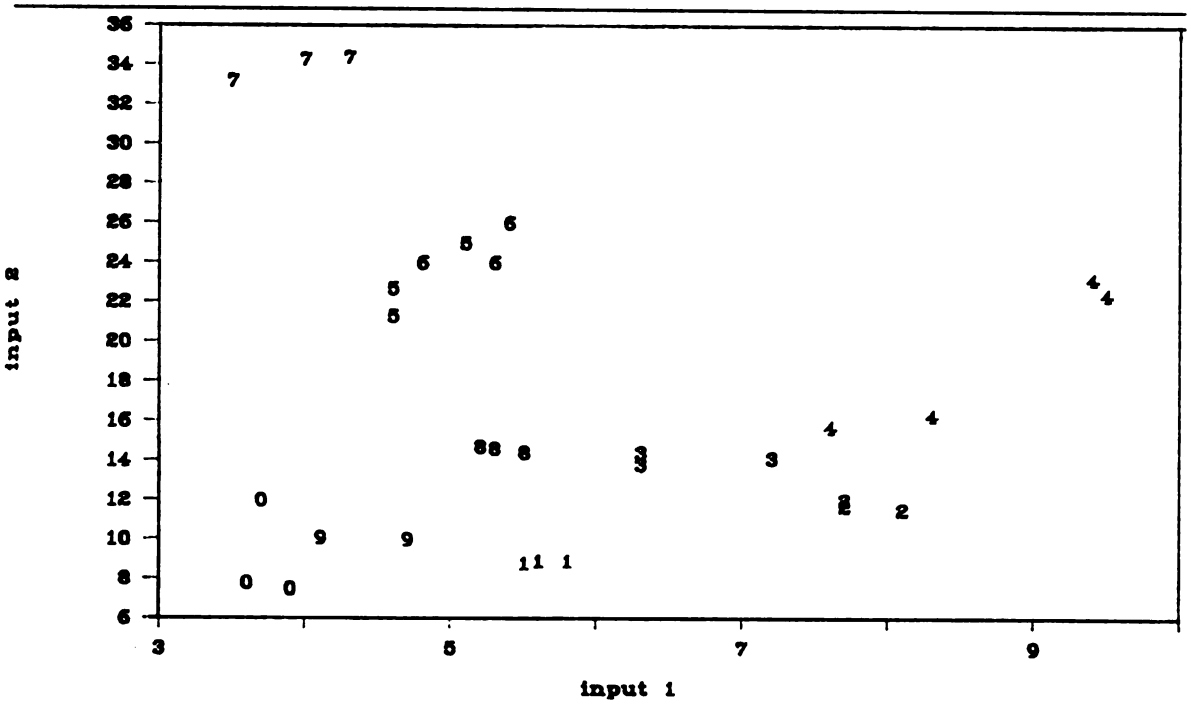

Figure 4.33: Training set N.

Table 4.8: Iterations to convergence of the BP algorithm for various training sets and varying number of hidden layer nodes.

	Number of hidden layer nodes					
Training set	2	3	4	5	6	7
A	71	95	84			
B	74	84	79			
C	66	73	75			
D	66	73	74			
E	163	75	76			
F	163	74	76			
G	-	-	-			
H	-	-	-			
I	-	-	-			
J	-	-	-			
K	-	-	-			
L	192	295	306			
M	-	160	165			
O				-	-	-

lowest error and the most correctly classified training patterns. Table 4.12 shows the number of correctly classified training patterns for different number of hidden layer nodes. Figure 4.35 shows the relative errors of the BP algorithm using six, seven, and eight hidden layer nodes. Figures 4.36 though 4.37 show the errors for the node-wise and layer-wise algorithms.

After performing these comparative simulations, it is possible to enumerate some of the advantages of the layer-wise algorithm over BP and the node-wise algorithm. First, two problems that arise in training, selection of initial weights and forgetting factor have been made easier. The layer-wise algorithm is relatively impervious to changes in these two variables, and they can be chosen small with very good results. Next, the layer-wise algorithm has very good convergence results with relatively complicated training sets. When selecting a training algorithm, many things need to be considered. Most important are speed and convergence performance. If a training

Table 4.9: Iterations to convergence of the node-wise algorithm for various training sets and varying number of hidden layer nodes.

Training set	Number of hidden layer nodes					
	2	3	4	5	6	7
A	3	5	4			
B	4	4	4			
C	5	5	6			
D	4	6	4			
E	6	6	8			
F	3	5	5			
G	467	326	380			
H	460	311	392			
I	-	-	664			
J	-	-	664			
K	-	-	-			
L	14	18	13			
M	241	103	157			
N				-	-	-

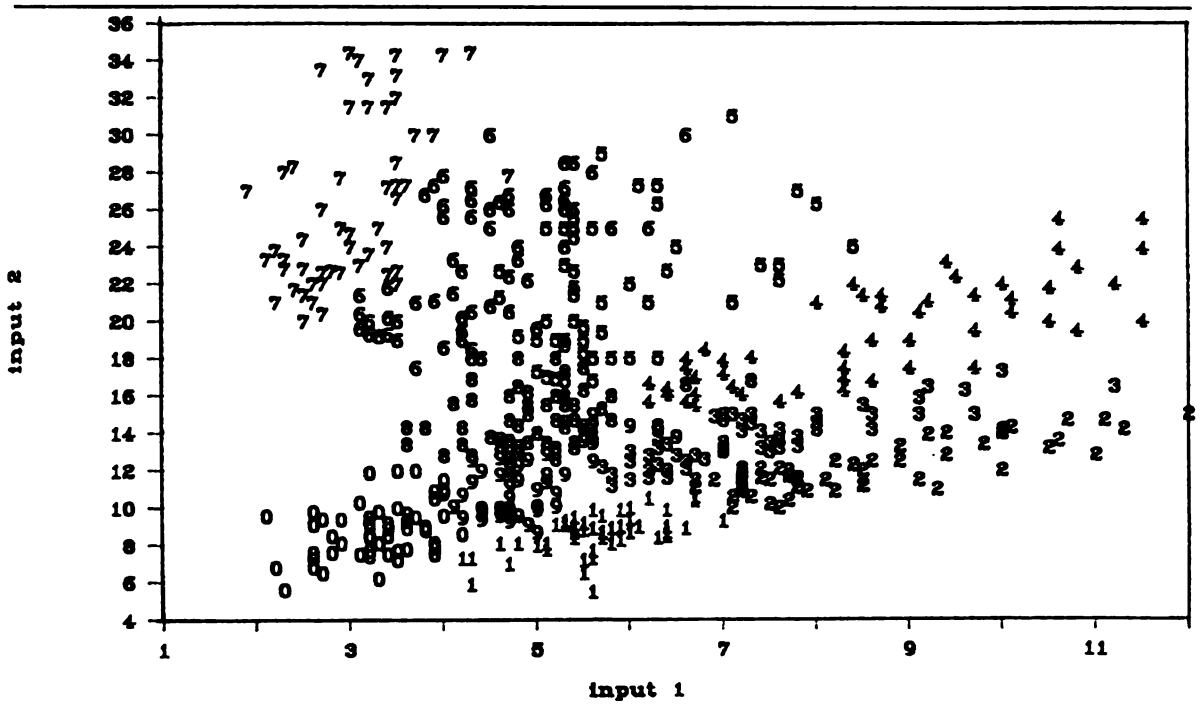

Figure 4.34: Large training set.

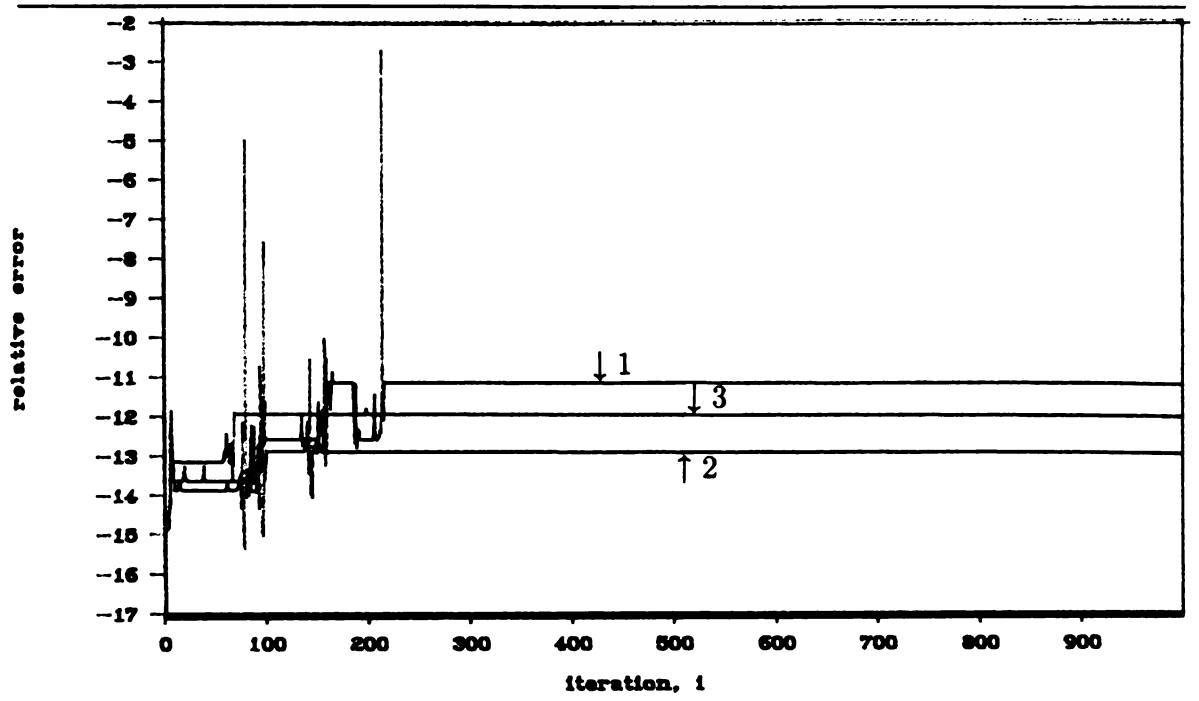

Figure 4.35: Average error vs. iteration number for the BP implementation, training with the large training set. 1. 6 nodes; 2. 7 nodes; 3. 8 nodes.

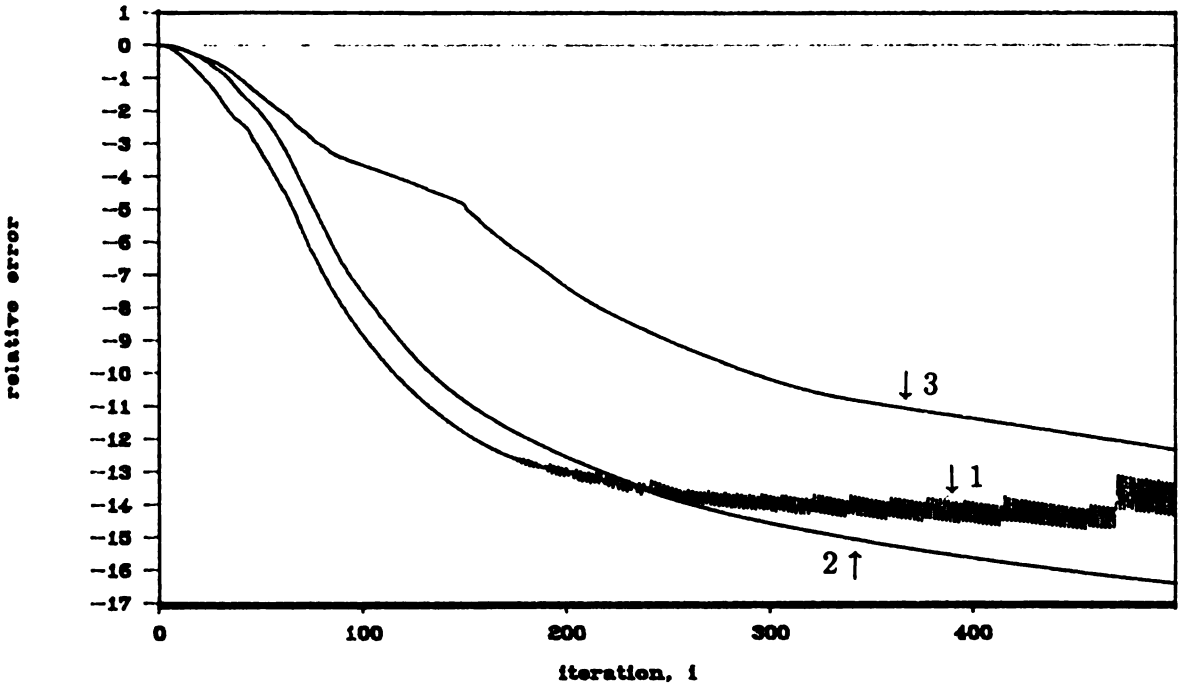

Figure 4.36: Average error vs. iteration number for the Node-wise implementation, training with the large training set. 1. 6 nodes; 2. 7 nodes; 3. 8 nodes.

Table 4.10: Iterations to convergence of the layer-wise algorithm for various training sets and varying number of hidden layer nodes.

Training set	Number of hidden layer nodes					
	2	3	4	5	6	7
A	19	6	18			
B	27	21	22			
C	24	9	23			
D	26	14	32			
E	25	19	23			
F	21	14	43			
G	118	34	30			
H	118	42	24			
I	-	17	27			
J	-	17	27			
K	-	133	28			
L	5	23	18			
M	71	76	117			
N				-	217	347

algorithm is very fast but rarely converges it is not good. Also, if the algorithm almost always converges but is very slow, then it is also not good. The true "speed" of a training algorithm must be determined by how fast it performs each iteration, how many iterations it takes on the average to converge, and how often it converges.

4.4 Simulations of the Data Reduction Algorithm

This section contains the results of simulations run using the data reduction algorithm. The data reduction algorithm was implemented along with the layer-wise training algorithm. The results compare the layer-wise algorithm with and without data reduction. The large data set introduced in the previous section is used for comparisons. Both algorithms are run using six, seven, and eight hidden layer nodes. The data reduction algorithm works by comparing the relative effect of the training

Table 4.11: Iterations to convergence of the network-wise algorithm for various training sets and varying number of hidden layer nodes.

Training set	Number of hidden layer nodes					
	2	3	4	5	6	7
A	-	-	65			
B	-	13	-			
C	34	-	-			
D	-	-	-			
E	-	-	-			
F	19	-	-			
G	-	-	-			
H	17	-	-			
I	-	-	-			
J	-	-	-			
K	-	-	-			
L						
M						
N						

Table 4.12: Number of correctly classified training patterns for the large training set for varying number of inner layer nodes.

Hidden layer Nodes	BP	Node-wise	Layer-wise
6	341	363	427
7	373	398	428
8	357	370	360

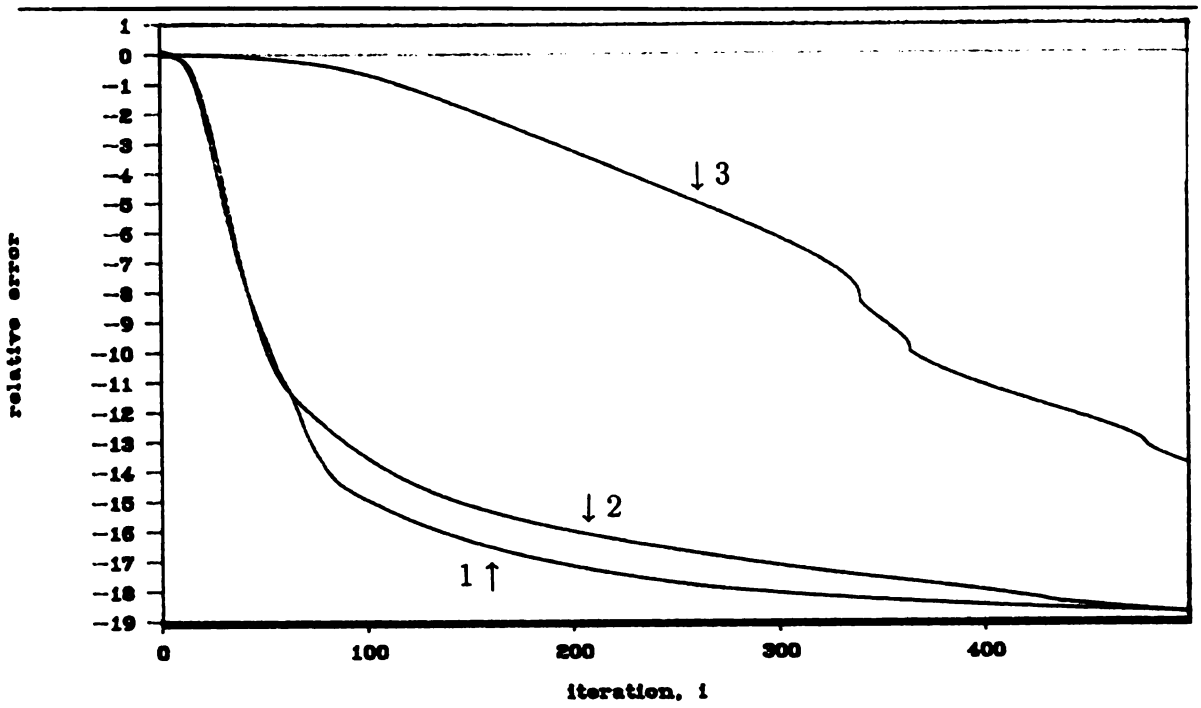

Figure 4.37: Average error vs. iteration number for the Layer-wise implementation, training with the large training set. 1. 6 nodes; 2. 7 nodes; 3. 8 nodes.

patterns on the magnitude of the weight change and does not use the training patterns with very small effects. Thus, for implementation the user must decide what is the level for accepting and rejecting training patterns for each iteration. The algorithm is implemented by computing the approximation to the upper bound on the weight change for each training pattern and selecting the largest of the approximations. This approximation is divided by a user defined factor to get a comparison number. All training patterns with approximations larger than this comparison number are used in the weight updating for the present iteration. This is then repeated for the next iteration. Figure 4.38 shows the relative error of three implementations of the layer-wise algorithm with six hidden layer nodes. The first has no data reduction. The second has data reduction and a dividing factor of 30. The third has data reduction and a dividing factor of 100. Figure 4.39 shows the number of correctly classified training patterns vs. iteration number for the same implementations. Figures 4.40

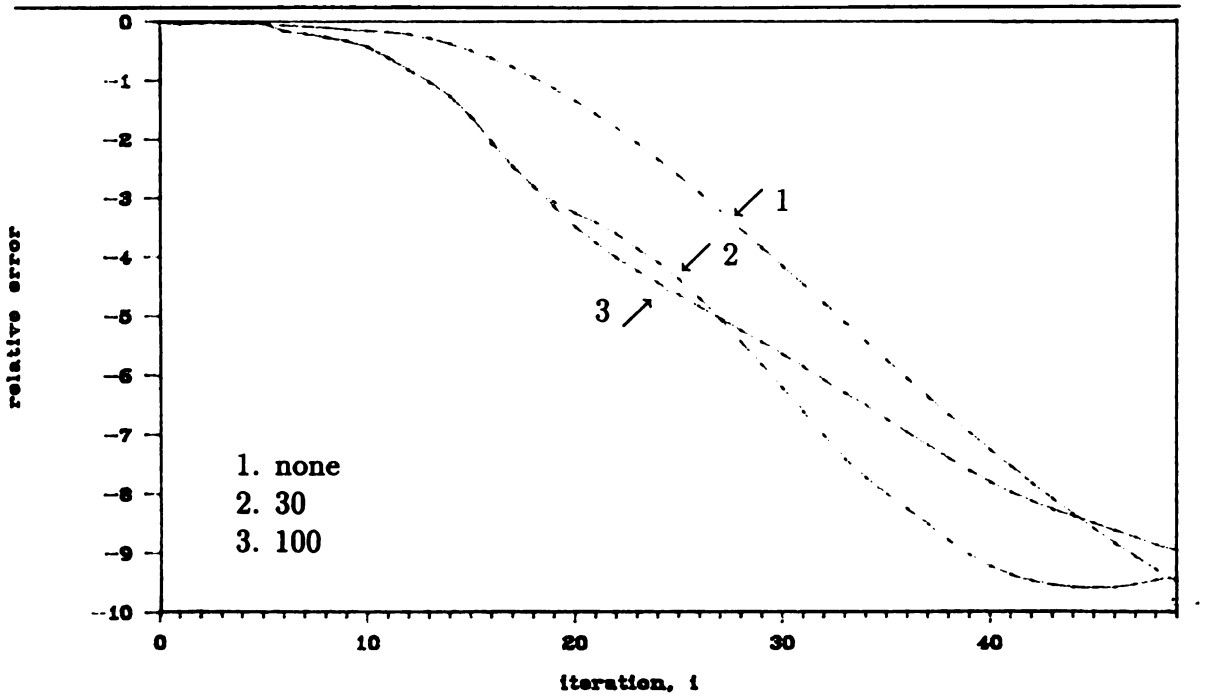

Figure 4.38: Average error vs. iteration number for the data reduction algorithm using 6 hidden layer nodes, training with the large training set.

and 4.41 are similar to above, showing the results for seven hidden layer nodes. Figures 4.42 and 4.43 show the results for eight hidden layer nodes. Figure 4.44 shows the number of training patterns used in the hidden layer vs. the iteration number for a dividing factor of 30 and varying hidden layer nodes. Figure 4.45 is similar, but a dividing factor of 100.

The results show the data reduction algorithm performing very well at the beginning, with the algorithm implemented without data reduction occasionally catching up at later iterations. The results can be misleading, because the data reduction algorithm is using fewer training patterns, thus performs each iteration faster than the algorithm with no data reduction. If the figures were to compare error vs. time, the data reduction algorithm would be superior. The results indicate that using data reduction is very good at speeding up the algorithm in the first iterations. The error of the data reduction algorithm can actually increase if the number of training

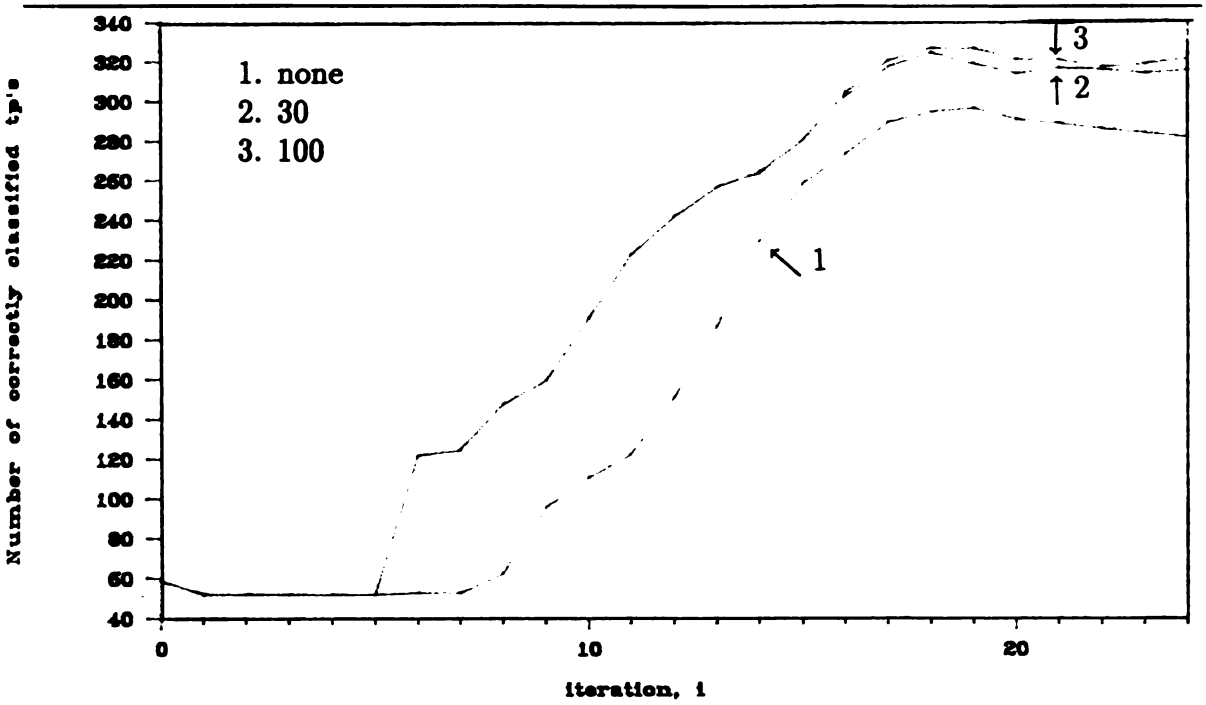

Figure 4.39: Correctly classified training patterns vs. iteration number for the data reduction algorithm using 6 hidden layer nodes and the large training set.

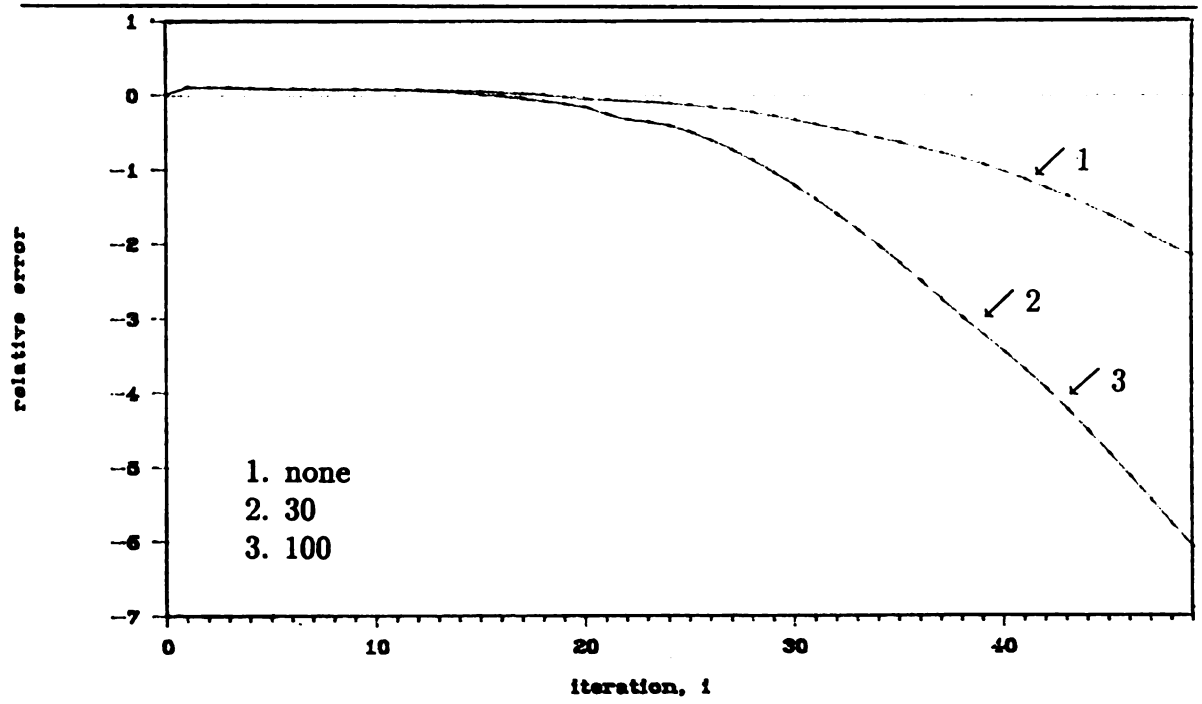

Figure 4.40: Average error vs. iteration number for the data reduction algorithm using 7 hidden layer nodes, training with the large training set.

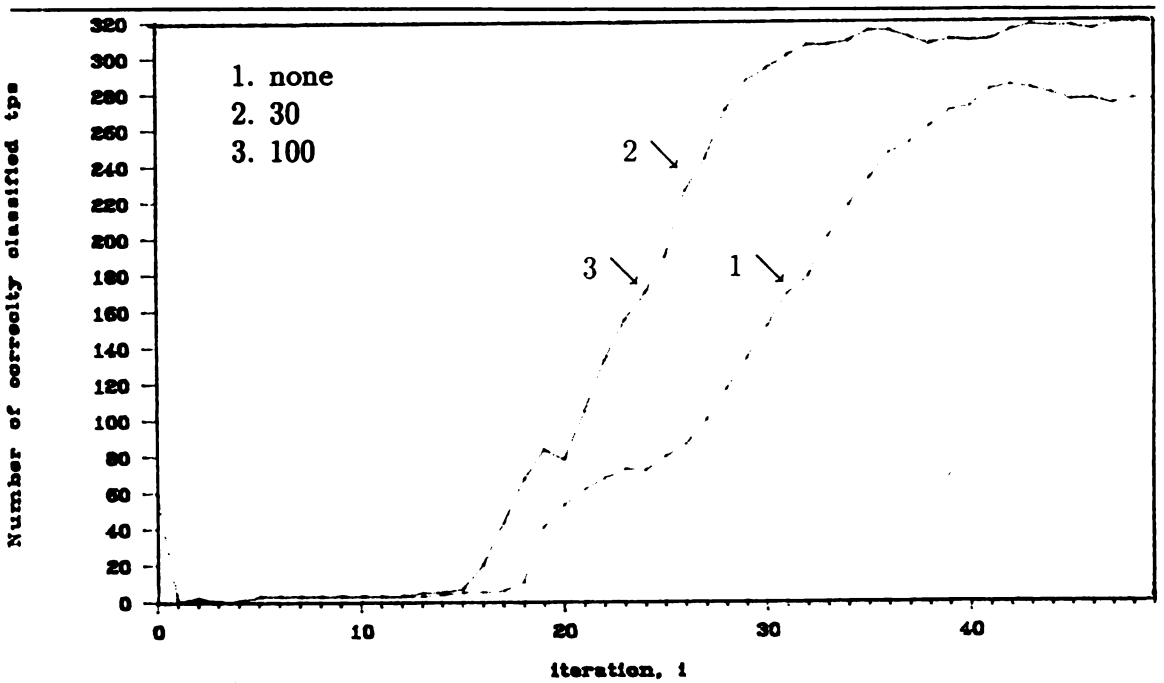

Figure 4.41: Correctly classified training patterns vs. iteration number for the data reduction algorithm using 7 hidden layer nodes and the large training set.

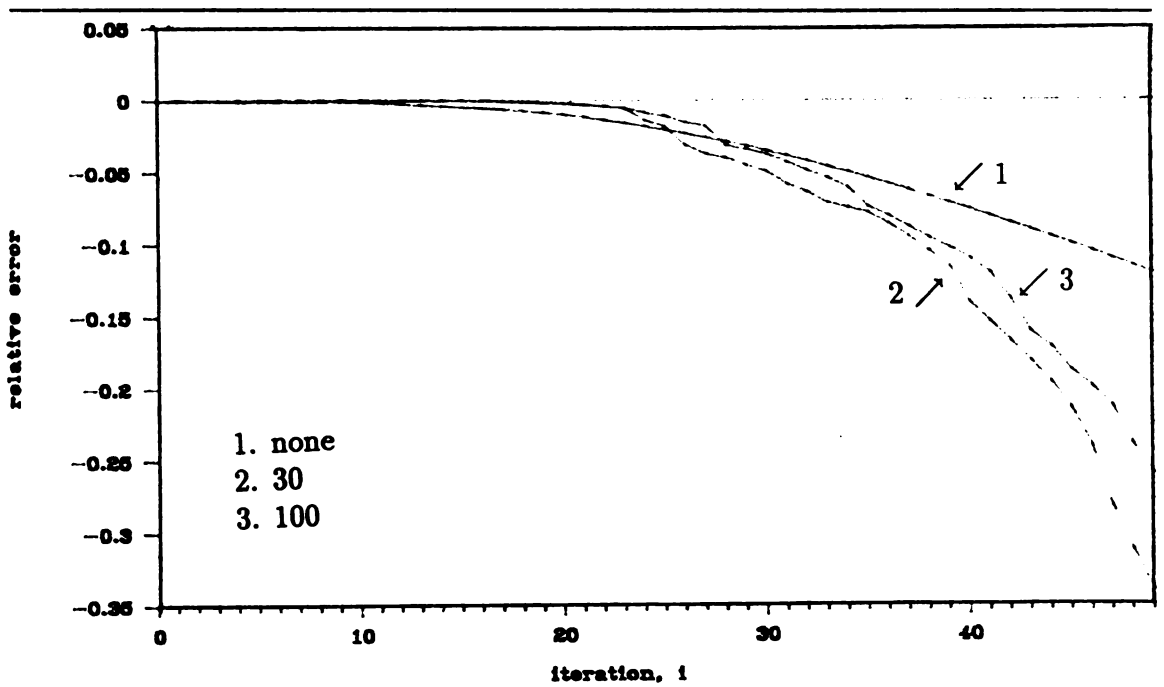

Figure 4.42: Average error vs. iteration number for the data reduction algorithm using 8 hidden layer nodes, training with the large training set.

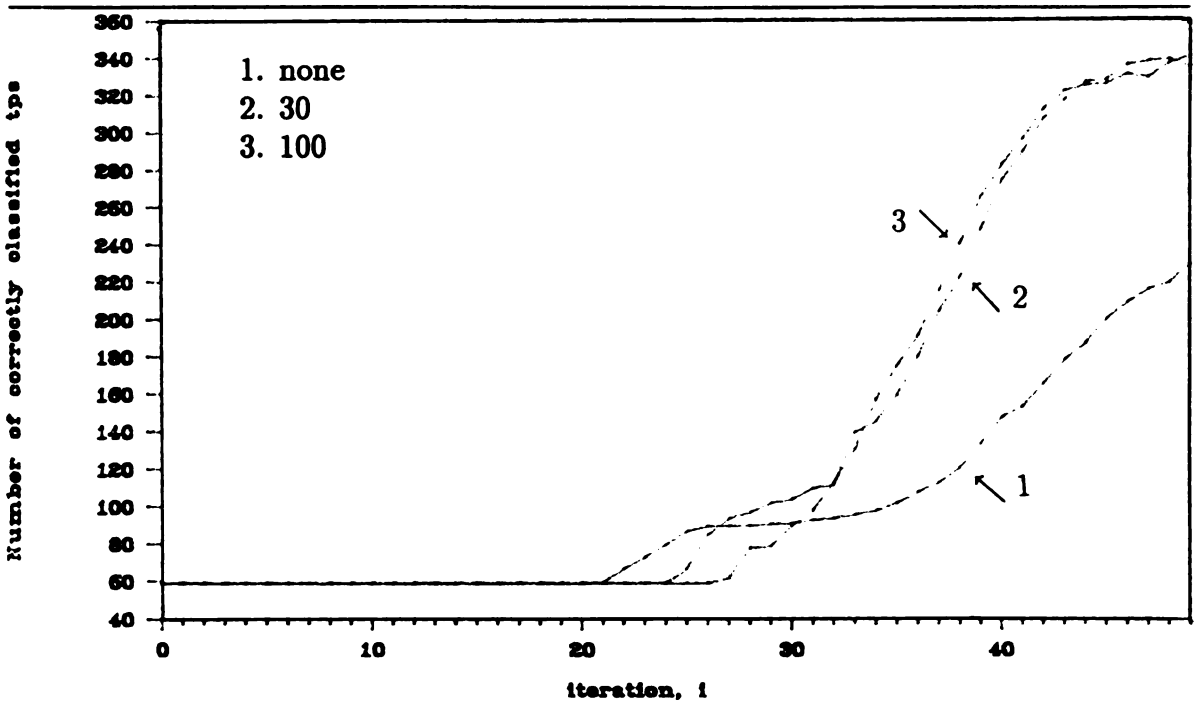

Figure 4.43: Correctly classified training patterns vs. iteration number for the data reduction algorithm using 8 hidden layer nodes and the large training set.

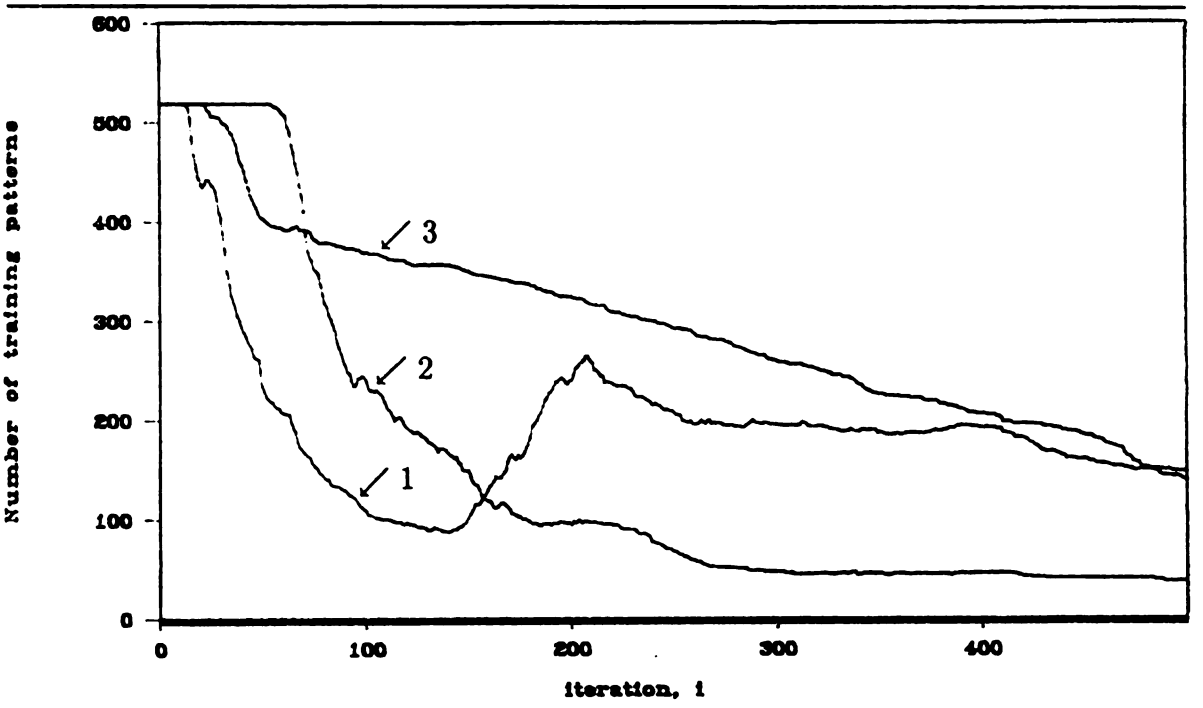

Figure 4.44: Number of training patterns used vs. iteration number for the data reduction algorithm and a dividing factor of 30. 1. 6 nodes 2. 7 nodes 3. 8 nodes.

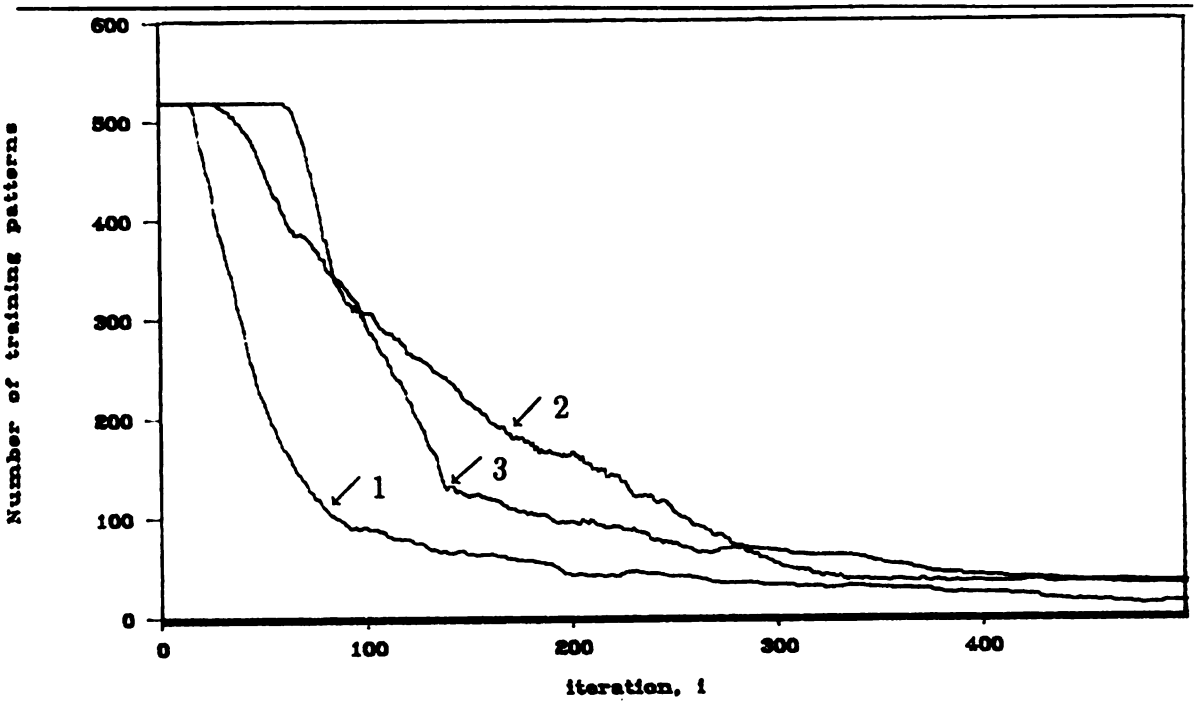

Figure 4.45: Number of training patterns used vs. iteration number for the data reduction algorithm and a dividing factor of 100. 1. 6 nodes 2. 7 nodes 3. 8 nodes.

patterns used becomes too small. In practice this number can be easily monitored and if the error increases, more training patterns can be used at the next iteration. The data reduction algorithm increases the algorithm efficiency considerably at the beginning of training because of the method of selecting the training patterns to be used in the updating. The training patterns that have the largest effect on the weight change are typically those that are near the center of the cluster of each class. If we divide the training pairs into classes, those that are near the center of the cluster, away from training pairs of different classes typically have the largest effect. These training pairs are then used to quickly find weights that classify the centers of the clusters correctly. If the class clusters are well separated, then the algorithm will do a good job of classifying all the training patterns. If there is overlapping among the clusters, then more training patterns must be used as training proceeds to refine the weights and classify the outliers correctly. The large training set used here has

different classes with a lot of overlapping, so is the most difficult challenge for the data reduction algorithm.

The number of training patterns used in the weight updating decreases as training proceeds. This indicates that all the training patterns have similar effect on the magnitude of the weight change at the beggining of training, with some having less and less effect as training progresses. This confirms the observation that began the investigation for a data reduction algorithm. The weight updating matrix used for QR-WRLS is largest for the hidden layer. This size is important, because increasing one weight greatly increases the number of rotations in the QR-WRLS algorithm. For example, with eight hidden layer nodes and three inputs, (as in the simulation here) there are 24 hidden layer weights. This means a 25 by 25 matrix must be used for updating. There are 314 rotations for each training pattern. With only one more weight in the hidden layer the number of rotations for each training patterns increases to 340. This means that for even modest sized FNNs, those havin six or more hidden layer nodes, the number of training patterns needed to remove, so that the data reduction algorithm is faster than the non-data reduction algorithm is small. For this example if, fewer than 92 percent of the training patterns are used, the data reduction algorithm is faster.

Chapter 5

Conclusions and Future Research

5.1 Algorithmic Developments

5.1.1 Training Speed

New algorithms to train FNNs are developed in this dissertation. The main purpose in developing new algorithms is to make training more efficient. Let us define the speed of an algorithm as the time it takes to converge to a solution. In measuring the speed of an algorithm, we need to know how fast it completes each iteration, how many iterations it takes, on the average, to converge, and how often it converges. Determining the speed of any of the algorithms used to train FNNs is quite difficult, if not impossible. This is due to several factors. First, algorithms are dependent on initial conditions. Thus for different initial conditions, the same algorithm may converge very fast, slowly or not at all. Next, the speed of an algorithms depends on the training set being used. This makes it difficult to compare training algorithms. Many initial conditions for the same training set, and many different training sets must be used to insure the comparisons are fair. The simulations done in Chapter 4 show that the layer-wise algorithm described in this dissertation is indeed an improvement over previous algorithms. As can be seen from the simulations, if the training set is very simple, and linearly separable, then any of the training algorithms can be used

to train the FNN. If the training set is more complicated, however, then usin the layer-wise algorithm greatly enhances performance. One way to speed up training is to decrease the average number of iterations to a solution and to increase the number of convergences. Another way of speeding up training is to decrease the time for each iteration. The layer-wise algorithm uses the previous method. The data reduction algorithm uses the latter method

5.1.2 Layer-wise Training Algorithm

The layer-wise training algorithm can be viewed as a logical progression from previous training methods. The BP algorithms takes the derivative of the error surface with respect to one weight, and changes this weight to reduce the total output error. Thus each weight is changed individually to decrease the error. Next, many algorithms, such as the A-S algorithm, and the algorithm developed by Kollias and Anastasiou take the derivative of the weights connected to each node, and change the weights to reduce the error. Here, all the weights connected to one node are changed together to decrease the error. In the layer-wise algorithm the derivative with respect to all the weights in the same layer is taken, and these weight are changed to reduce the error. The layer-wise algorithm is a linear algorithm in that the non-linearities of the FNN are linearized around the present weights at each iteration, and this linerized approximation is used to update the weights. This makes possible the use of QR-WRLS for solving the linear system at each iteration.

5.1.3 QR-WRLS

The use of derivatives of order greater than one in the training of FNNs has become popular. The use of higher order derivatives means that the system can be linerized so that the resulting equations are linear, and can be solved for many of the weights simultaneously using some RLS algorithm, typically MIL-WRLS. It was one of the

purposes of this dissertation to introduce the QR-WRLS algorithm for use in FNN training algorithms. Future algorithms introduced for the training of FNNs will benefit from implementation with QR-WRLS rather than MIL-WRLS.

5.1.4 Data Reduction Algorithm

In training the FNN it was found that not all the training patterns are useful at each iteration in updating the system weights. A computationally inexpensive way of computing whether a training pattern is useful or not for use in updating the weights at the present iteration is developed. This allowed fewer training patterns to be used at each iteration, speeding up training.

5.2 Future Research

The most exciting future research lies in the extension of the layer-wise algorithm. As mentioned above, training algorithms have developed from updating one weight at a time, to all the weights connected to one node at a time, to all the weights of the same layer at a time. The next step will be updating all the weights of the network simultaneously. The network-wise algorithm presented in Chapter 3 is a first step in this direction. It is limited however, in that only one output is allowed. Also, as noted in Chapter 4 the performance is not as good as the layer-wise algorithm. To update all the weights of the same layer, only a first order liner approximation to the FNN is needed. In updating all the weights of the network, a higher order approximation may be used.

The set membership algorithm [31],[32] was the first algorithm used in an attempt to reduce the number of training patterns used at each iteration. This algorithms can be used in linear systems when there is a bound on the output error. The bounded output of the FNN made it seem a natural problem for the application of

set membership. It was quickly noted, however, that although the bounded output of the FNN does imply a bound on the error, it does not imply a bound on the solution weights. This occurs because of the non-linearities in the system. It was because of this that perturbation theory was used. However, set membership theory in another form may still be applicable to the FNN training problem, and this application is a topic of future reseach.

Bibliography

[1] T. Sejnowski, and C. Rosenberg, "Nettalk: a parallel network that leans to read aloud," Technical Report JHU/EECS-86/01, Johns Hopkins University, 1986.

[2] D. Nguyen, and B. Widrow, "The truck backer-upper: and example of self-learning in neural networks." *Proceedings of the IEEE Int. Joint Conf. on Neural Networks*, pp.1381-1384, 1990.

[3] A.S. Householder, "The approximate solution of matrix problems," *J. Assoc. Comput. Mach.*, vol.5, pp. 204-243,1958.

[4] J.R. Deller, Jr., and D. Hsu, "An alternative adaptive sequential regression algorithm and its application to the recognition of cerebral palsy speech," *IEEE Transactions on Circuits and Systems*, vol.CAS-34, no.7, pp.782-786, July 1987

.

[5] D. Graupe, *Time Series Analysis, Identification and Adaptive Filtering*, Malabar, Florida: Krieger Publishing Company, 1989.

[6] M. Azimi-Sadjadi, S. Citrin, and S. Sheedvash, "Supervised learning process of multi-layer perceptron neural networks using fast least squares," *Proceedings of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, pp.1381-1384, 1990.

[7] G. Golub and C. VanLoan, *Matrix Computations*, Baltimore, Maryland: Johns Hopkins University Press, 1983.

[8] J. Sun, W. Grosky, and M. Hassoun, "A fast algorithm for finding global minima of error functions in layered neural networks," *Proceedings of the IEEE Int. Joint Conf. on Neural Networks*, vol.1, pp.715-720, 1990.

[9] B. Widrow, and M. Lehr, "30 years of adaptive neural networks: perception, madeline and backpropagation," *Proceedings of the IEEE* , vol.78, no.9, September 1990.

[10] F. Rosenblatt, *Principles of neurodynamics: perceptrons and the theory of brain mechanisms,* Washington, D.C.,: Spartan Books, 1962.

[11] R. Watrous, "Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization," *Proceedings of the IEEE Int. Joint Conf. on Neural Networks*, vol.2, pp.619-627, 1987.

[12] D. Rumelhart, G. Hinton, and G. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*, vol.1 (D. Rumelhart and J. McCleland, Eds.). Cambridge, MA: MIT Press, 1986.

[13] R. Lippman, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol.4, pp.4-22, Apr.1987.

[14] J.R. Deller, Jr., J. Proakis, and H. Hansen, *Discrete Time Processing of Speech Signals*, 1992.

[15] L. Johnson, R. Riess, *Numerical Analysis*, 2nd ed., Reading, MA. Addison-Wesley Pub.Co., 1982.

[16] D. Parker, "Learning logic," Technical Report TR-40, Center for Computational Research in Economics and Management Science, MIT, April, 1985.

[17] P. Werbos, "Beyond regression: New tool for prediction and analyses in the behavioral sciences," *Ph.D. dissertation*, Harvard Univ., Cambridge, MA, 1974.

[18] F.M.A. Salam, "Neural nets and engineering implementations," *Key Address, The Proc. of the 31st Midwest Symposium on Circuits and Systems*, St.Louis, MO., Aug. 1988

[19] M. Azimi-Sadjadi and S. Citrin, "Fast leaning process of multi- layer neural nets using recursive least squares technique," *IEEE Int. Conf. on Neural Networks*, Washington D.C., June 1989.

[20] E. Dahl, "Accelerated learninig using the generalized delta rule," *Proceedings of the IEEE Int. Joint Conf. on Neural Networks*, vol.2, pp.523-530, 1987.

[21] R. White, "The learning rate in back-propagation systems: an application of newtons method," *Proceedings of the IEEE Int. Joint Conf. on Neural Networks*, vol.1, pp.679-684, 1990.

[22] D. Parker, "Second order backpropagation: Implementing an optimal $\mathcal{O}(n)$ approximation to Newton's method as an artificial neural network," *IEEE Conf. on Neural Information Processing Systems*, Denver, CO, Nov. 1987.

[23] S. Kollias and D. Anastassiou, "An adaptive least squares algorithm for the efficient training of artificial neural networks," *IEEE Transactions on Circuits and Systems*, vol. CAS-36, pp.1092-1101, August 1989.

[24] J. Steck, B. McMillin, K. Krishnamurthy, M. Ashouri, and G. Leininger, "Parallel implementation of recursive least squares neural network training method,"

Proceedings of the IEEE Int. Joint Conf. on Neural Networks, vol.1, pp.631-636, 1990.

[25] T. Ghiselli-Crippa, and A. El-Jaroudi, "Afast neural net training algorithms and its application to voiced-unvoiced-silence classification of speech," *Proceedings of the IEEE Int. Joint Conf. on Neural Networks*, vol.1, pp.441-444, 1991.

[26] B. Kosko, *Neural Networks and Fuzzy Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1992.

[27] B. Widrow and S.D. Sterns, *Adaptive Signal Processing*, Englewood Cliffs, New Jersey: Prentice-Hall, 1985.

[28] J.R. Deller, Jr., and G. Picache, "Advantages of a givens rotation approach to temporally recursive linear prediction analysis of speech," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, pp. 429- 431, March 1989.

[29] L. Allred, "Supervised learning techniques for backpropagation networks," *Proceedings of the IEEE Int. Joint Conf. on Neural Networks*, vol.1, pp.721-728, 1990.

[30] D. Watkins, *Fundamentals of matrix computations*, New York, N.Y.,John Wiley and Sons, 1991.

[31] J.R. Deller, Jr. and T.C. Luk, "Linear prediction analysis of speech based on set-membership theory," *Computer Speech and Language*, vol. 3, pp.301-327,1989.

[32] J.R. Deller, Jr., "Set membership identification in digital signal processing," *IEEE ASSP Magazine*, pp.4-20, October 1989.

Appendix

Appendix

Listings of Training Programs

This Appendix lists the programs developed in this dissertation. The first program listed is the node-wise algorithm. Next the layer-wise algorithm is listed. The network-wise algorithm follows. The last program listed is the layer-wise algorithm with data reduction. All programs implement learning for a two layer network. All programs were written in a similar manner. The main program inputs the training set, and user defined variables such as the number of hidden layer nodes and the number of iterations. The number of input and output nodes is determined by the training set. Here also the weights are initialized along with the other algorithm variables such as the forgetting factor and weight constraint. AS2(), the main subroutine in each program, keeps track of the iteration number, calculates the outputs of the network, then uses these values to calculate the linearized training values. The first main subroutine is the rotein() subroutine. This routine rotates the linearized training pairs into the correct W matrix. There is one W matrix for each node in the node-wise algorithm. In the layer-wise algorithm there is one W matrix for each node in the output layer, and one W matrix for the hidden layer. The network-wise algorithm only uses one W matrix. The next main subroutine is weightch(). This subroutine uses the W matrix to calculate the updated weights after every iteration. The other two smaller subroutines are ASw() and initw(). ASw() multiplies the W

matrices by a forgetting factor. initw() initializes the W matrices to zero. The only differences between the node-wise, layer-wise, and network-wise algorithm is in how the AS2() subroutine calculates the linearized training patterns. The layer-wise algorithm with data reduction is the same as the layer-wise algorithm with the data reduction included. The first change is in the AS2() subroutine. Here for simplicity of programming the outputs of the network are calculated twice. First when using the data reduction algorithm to decide which training patterns to use, then when using these training patterns to update the W matrices. There are three additional subroutines. The first is the reinitwpp() subroutine. This initializes the Wpp matrix, which is used in determining which training patterns to use in the updating, to zero. The next subroutine is called norms(). This subroutine calculates the norms for the weights, the $\boldsymbol{w}(i)$ vectors and for the $\boldsymbol{d}_1(i, N)$, here called b vectors. The final subroutine is called dbdA(), and calculates the estimate of the weight change vector norm.

The input to all the programs is a training set with the following format;

number of training patterns,number of inputs,number of outputs

training pattern 1

training pattern 2

.

.

.

training pattern N

At each iteration the program writes to the screen the iteration number, the total number of training patterns, the number of correctly classified training patterns, and the sum of the squared errors between the training pattern outputs and the actual outputs.

Node-wise Algorithm

```
#include¡stdio.h¿
#include¡math.h¿
#include¡stdlib.h¿

#define Nlevels 2
/*
This program simulates a N0-input, N2-output 2-layer, feedforward
neural net using the Azimi-Sadjadi algorithm with G-R and
includes biasing. node-wise updating.
*/

int datapts,xdata,ydata,numweight,zed,use,Nloops,N[3];
float x[4][519],t[4][519];
float w[3][10][10],wa[30];
float K[3][10],b[3][10],delta1,delta2;
float W[13][10][10];
float y[3][10],wchange,err1;
float z[30],tp;
float ASweight;
double err/*,E[1000]*/;
char outfile[40];
FILE *ifp1,*ofp1,*ofp2,*ofp3;

main()
{
int k1,k2,k3;
void AS2();


printf("Enter name for input file: ");
gets(outfile);
if((ifp1=fopen(outfile,"r"))==NULL)
{
printf("fopen failed\n");
exit(0);
}


printf("Enter name for output file1: ");
gets(outfile);
if((ofp1=fopen(outfile,"w"))==NULL)
```

```
{
printf("fopen failed\n");
exit(0);
}


printf("Enter number of hidden nodes: ");
scanf("%d",&N[1]);

Here the input training set is read.
fscanf(ifp1,"%d,%d,%d\n",&datapts,&xdata,&ydata);
N[0]=xdata;
N[2]=ydata;
printf("datapts=%d xdata=%d ydata=%d\n",
datapts,xdata,ydata);
for(k1=0;k1¡datapts;++k1){
for(k2=0;k2¡xdata;++k2){
fscanf(ifp1,"%f,",&x[k2][k1]);
}
for(k2=0;k2¡ydata;++k2){
fscanf(ifp1,"%f,",&t[k2][k1]);
}
fscanf(ifp1,"\n");
}

printf("Enter maximum number of loops: ");
scanf("%d",&Nloops);
ASweight=0.1;
y[1][N[1]]=1;
y[0][N[0]]=1;
for(k1=0;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1]+1;++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
w[k1][k2][k3]=0.001;
}
}
}

reinitw();
wchange=0.10;

AS2();

fprintf(ofp1,"%d,%d,%d\n",N[0],N[1],N[2]);
```

```
/*for(k1=1;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1];++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
fprintf(ofp1,"%f\n",w[k1][k2][k3]);
printf("w[%d][%d][%d]=%f\n",
k1,k2,k3,w[k1][k2][k3]);
}
}
}*/

fclose(ifp1);
fclose(ofp1);
}

void AS2()
{
int n1,n2,n3,count,loop,data,test;

for(loop=0;loop¡Nloops;++loop){
zed=0;
err=0;
use=0;
ASw();
for(data=0;data¡datapts;++data){
for(n1=0;n1¡N[0];++n1){
y[0][n1]=x[n1][data];
}
Below the outputs of the nodes and the derivatives are calculated.
for(n1=1;n1¡Nlevels+1;++n1){
for(n2=0;n2¡N[n1];++n2){
y[n1][n2]=0;
for(n3=0;n3¡N[n1-1]+1;++n3){
y[n1][n2]+=y[n1-1][n3]*w[n1][n2][n3];
}
if(y[n1][n2]¡-20){y[n1][n2]=-20;}
b[n1][n2]=y[n1][n2];
y[n1][n2]=1/(1+exp(-y[n1][n2]));
K[n1][n2]=y[n1][n2]*(1-y[n1][n2]);
b[n1][n2]=y[n1][n2]-K[n1][n2]*b[n1][n2];
}
}
Here the linearized intputs and outputs of the last layer
are calculated.
for(n1=0;n1¡N[Nlevels];++n1){
if(t[n1][data]==0){tp=0.05-b[Nlevels][n1];}
```

```
else if(t[n1][data]==1){tp=0.95-b[Nlevels][n1];}
else{tp=t[n1][data]-b[Nlevels][n1];}
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
z[n2]=K[Nlevels][n1]*y[Nlevels-1][n2];
}
rotin(N[Nlevels-1]+1,n1+1);
}
```
Below the linearized inputs and outputs of the hidden layer are calculated.
```
for(n2=0;n2¡N[Nlevels-1];++n2){
tp=0;
for(n1=0;n1¡N[Nlevels];++n1){
tp+=(t[n1][data]-y[Nlevels][n1])*K[Nlevels][n1]
*w[Nlevels][n1][n2];
}
tp+=y[Nlevels-1][n2];
tp=tp-b[Nlevels-1][n2];
for(n3=0;n3¡N[Nlevels-2]+1;++n3){
z[n3]=K[Nlevels-1][n2]*y[Nlevels-2][n3];
}
rotin(N[Nlevels-2]+1,n2+N[Nlevels]+1);
}
```
The sum of square errors is calculated below.
```
for(n1=0,err1=0,test=0;n1¡ydata;++n1){
err1+=(t[n1][data]-y[2][n1])*(t[n1][data]-y[2][n1]);
if(fabs(t[n1][data]-y[2][n1]) ¿= 0.5){test=1;}
}
if(test==0){zed++;}
err+=err1;
}
```
Below the new weights for the last layer are calculated.
```
for(n1=0;n1¡N[Nlevels];++n1){
weightch(Nlevels,N[Nlevels-1]+1,n1+1);
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
w[Nlevels][n1][n2]=wa[n2];
}
}
```
Below the new weights for the hidden layer are calculated.
```
for(n2=0;n2¡N[Nlevels-1];++n2){
weightch(Nlevels-1,N[Nlevels-2]+1,n2+N[Nlevels]+1);
for(n3=0;n3¡N[Nlevels-2]+1;++n3){
w[Nlevels-1][n2][n3]=wa[n3];
}
}
printf("loop=%d data=%d zed=%d err[data]=%f\n",
```

```
loop,data,zed,err);
fprintf(ofp1,"loop=%d zed=%d err=%f\n",loop,zed,err);
}
}
```

The following subroutine rotates the linearized training patterns
into the W matrices using QR decomposition.

```
rotin(MATSIZE,mat)
int MATSIZE,mat;
{
int k1,k2;
float rho,sigma,tau,Wp[30][30];

for(k2=0;k2¡MATSIZE;++k2){
W[mat][MATSIZE][k2]=z[k2];
}
W[mat][MATSIZE][MATSIZE]=tp;

for(k2=0;k2¡MATSIZE+1;++k2){
for(k1=0;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=0;
}
}
for(k2=0;k2¡MATSIZE;++k2){
rho=sqrt(W[mat][k2][k2]*W[mat][k2][k2]+
W[mat][MATSIZE][k2]*W[mat][MATSIZE][k2]);
if(rho == 0){/*if(mat==0){printf("below\n");
}*/goto below4;}
sigma=W[mat][k2][k2]/rho;
tau=W[mat][MATSIZE][k2]/rho;
for(k1=k2;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=W[mat][k2][k1]*sigma+W[mat][MATSIZE][k1]*tau;
Wp[MATSIZE][k1]=-W[mat][k2][k1]*tau+W[mat][MATSIZE][k1]*sigma;
}
Wp[MATSIZE][k2]=0;
for(k1=0;k1¡MATSIZE+1;++k1){
W[mat][k2][k1]=Wp[k2][k1];
W[mat][MATSIZE][k1]=Wp[MATSIZE][k1];
}
below4:;
}

}
```

The following subroutine re-initializes the W matrices.

```
reinitw()
{
int n1,n2,n3;
for(n1=0;n1¡7;++n1){
for(n2=0;n2¡30;++n2){
for(n3=0;n3¡30;++n3){
W[n1][n2][n3]=0;
}
}
}

}
```

The following subroutine uses the W matrices to calculate
the updated weights.

```
weightch(level,MATSIZE,mat)
int level,MATSIZE,mat;
{
double wchangep[30],a[30],zzt,zt,wp[30],den,fac;
int k,i,n1,n2,n3,counts;

zt=0;

if(level==2){
for(n1=0;n1¡MATSIZE;++n1){
wp[n1]=w[level][mat-1][n1];
}
}
if(level==1){
for(n1=0;n1¡MATSIZE;++n1){
wp[n1]=w[level][mat-N[Nlevels]-1][n1];
}
}

if(fabs(W[mat][MATSIZE-1][MATSIZE-1])¿1e-28){
a[MATSIZE-1]=W[mat][MATSIZE-1][MATSIZE]/W[mat][MATSIZE-1][MATSIZE-1];
}
else{
printf("no weight update level %d %d\n",level,MATSIZE);
a[MATSIZE-1]=wp[MATSIZE-1];
}
for(n1=MATSIZE-2;n1¿=0;-n1){
if(fabs(W[mat][n1][n1])¿1e-28){
a[n1]=W[mat][n1][MATSIZE];
for(n2=MATSIZE-1;n2¿n1;-n2){
```

```
a[n1]-=W[mat][n1][n2]*a[n2];
}
a[n1]/=W[mat][n1][n1];
}
else{
printf("no weight update level %d %d\n",level,MATSIZE);
a[MATSIZE-1]=wp[MATSIZE-1];
}
}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]=a[n1]-wp[n1];
}
den=0;
for(n1=0;n1¡MATSIZE;++n1){
den+=wchangep[n1]*wchangep[n1];
}
den=sqrt(den);
if( den ¿ wchange){fac=wchange/den;}
else{fac=1;}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]*=fac;
}
for(n1=0;n1¡MATSIZE;++n1){
wa[n1]=wp[n1]+wchangep[n1];
}
for(n1=0,zt=0;n1¡MATSIZE;++n1){
if(wa[n1]¿700){wa[n1]=700;zt=1;}
if(wa[n1]¡-700){wa[n1]=-700;zt=1;}
}

if(zt==1){printf("weight at 700\n");}
}
```

The following subroutine implements the forgetting factor by multiplyint the W matrices by the constant ASweight.

```
ASw()
{
int n1,n2,n3;

for(n1=0;n1¡10;++n1){
for(n2=0;n2¡10;++n2){
for(n3=0;n3¡10;++n3){
W[n1][n2][n3]*=ASweight;
}
}
```

```
}
}
```

Layer-wise Algorithm

```
#include¡stdio.h¿
#include¡math.h¿
#include¡stdlib.h¿
#define Nlevels 2
/*
This program simulates a N0-input, N2-output 2-layer, feedforward
neural net using the Azimi-Sadjadi algorithm with G-R and
includes biasing.
*/

int datapts,xdata,ydata,numweight,zed,use,Nloops,N[3];
float x[4][519],t[4][519];
float w[3][10][10],wa[30];
float K[3][10],b[3][10],delta1,delta2;
float W[7][30][30];
float y[3][10],wchange,err1;
float z[30],tp;
float ASweight;
double err/*,E[1000]*/;
char outfile[40];
FILE *ifp1,*ofp1,*ofp2,*ofp3;

main()
{
int k1,k2,k3;
void AS2();

printf("Enter name for input file: ");
gets(outfile);
if((ifp1=fopen(outfile,"r"))==NULL)
{
printf("fopen failed\n");
exit(0);
}

/*printf("Enter name for output file1: ");
gets(outfile);
if((ofp1=fopen(outfile,"w"))==NULL)
{
printf("fopen failed\n");
exit(0);
```

```
}*/

printf("Enter number of hidden nodes: ");
scanf("%d",&N[1]);

Here the input training set is read.
fscanf(ifp1,"%d,%d,%d\n",&datapts,&xdata,&ydata);
N[0]=xdata;
N[2]=ydata;
printf("datapts=%d xdata=%d ydata=%d\n",
datapts,xdata,ydata);
for(k1=0;k1¡datapts;++k1){
for(k2=0;k2¡xdata;++k2){
fscanf(ifp1,"%f,",&x[k2][k1]);
}
for(k2=0;k2¡ydata;++k2){
fscanf(ifp1,"%f,",&t[k2][k1]);
}
fscanf(ifp1,"\n");
}

printf("Enter maximum number of loops: ");
scanf("%d",&Nloops);
ASweight=0.1;
Below the weights are inintialized.
y[1][N[1]]=1;
y[0][N[0]]=1;
for(k1=0;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1]+1;++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
w[k1][k2][k3]=0.001;
}
}
}

reinitw();
wchange=.4;

AS2();

/*fprintf(ofp1,"%d,%d,%d\n",N[0],N[1],N[2]);
for(k1=1;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1];++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
fprintf(ofp1,"%f\n",w[k1][k2][k3]);
```

```
printf("w[%d][%d][%d]=%f\n",
k1,k2,k3,w[k1][k2][k3]);
}
}
}*/

fclose(ifp1);
fclose(ofp1);
}
void AS2()
{
int n1,n2,n3,count,loop,data,test;

for(loop=0;loop¡Nloops;++loop){
zed=0;
err=0;
use=0;
ASw();
Below the outputs of the nodes and the derivatives are calculated.
for(data=0;data¡datapts;++data){
for(n1=0;n1¡N[0];++n1){
y[0][n1]=x[n1][data];
}
for(n1=1;n1¡Nlevels+1;++n1){
for(n2=0;n2¡N[n1];++n2){
y[n1][n2]=0;
for(n3=0;n3¡N[n1-1]+1;++n3){
y[n1][n2]+=y[n1-1][n3]*w[n1][n2][n3];
}
f(y[n1][n2]¡-20){y[n1][n2]=-20;}
b[n1][n2]=y[n1][n2];
y[n1][n2]=1/(1+exp(-y[n1][n2]));
K[n1][n2]=y[n1][n2]*(1-y[n1][n2]);
b[n1][n2]=y[n1][n2]-K[n1][n2]*b[n1][n2];
}
}
Here the linearized intputs and outputs of the last
layer are calculated.
for(n1=0;n1¡N[Nlevels];++n1){
tp=t[n1][data]-b[Nlevels][n1];
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
z[n2]=K[Nlevels][n1]*y[Nlevels-1][n2];
}
rotin(N[Nlevels-1]+1,n1+1);
}
```

```
Below the linearized inputs and outputs of the hidden layer
are calculated.
for(n1=0;n1¡N[Nlevels];++n1){
count=0;
tp=t[n1][data]-b[Nlevels][n1];
for(n2=0;n2¡N[Nlevels-1];++n2){
tp-=K[Nlevels][n1]*w[Nlevels][n1][n2]*b[Nlevels-1][n2];
for(n3=0;n3¡N[Nlevels-2]+1;++n3){
z[count]=K[Nlevels][n1]*K[Nlevels-1][n2]*
w[Nlevels][n1][n2]*y[Nlevels-2][n3];
++count;
}
}
tp-=K[Nlevels][n1]*w[Nlevels][n1][n2]*y[Nlevels-1][n2];
rotin(N[Nlevels-1]*(N[Nlevels-2]+1),0);
}
The sum of square errors is calculated below.
for(n1=0,err1=0,test=0;n1¡ydata;++n1){
err1+=(t[n1][data]-y[2][n1])*(t[n1][data]-y[2][n1]);
if(fabs(t[n1][data]-y[2][n1]) ¿= 0.5){test=1;}
}
if(test==0){zed++;}
err+=err1;
}
Below the new weights for the last layer are calculated.
for(n1=0;n1¡N[Nlevels];++n1){
weightch(Nlevels,N[Nlevels-1]+1,n1+1);
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
w[Nlevels][n1][n2]=wa[n2];
}
}
Below the new weights for the hidden layer are calculated.
weightch(Nlevels-1,N[Nlevels-1]*(N[Nlevels-2]+1),0);
count=0;
for(n1=0;n1¡N[Nlevels-1];++n1){
for(n2=0;n2¡N[Nlevels-2]+1;++n2){
w[Nlevels-1][n1][n2]=wa[count];
++count;
}
}
printf("loop=%d data=%d zed=%d err[data]=%f\n",
loop,data,zed,err);
}
}
```

The following subroutine rotates the linearized training patterns
into the W matrices using QR decomposition.

```
rotin(MATSIZE,mat)
int MATSIZE,mat;
{
int k1,k2;
float rho,sigma,tau,Wp[30][30];

for(k2=0;k2¡MATSIZE;++k2){
W[mat][MATSIZE][k2]=z[k2];
}
W[mat][MATSIZE][MATSIZE]=tp;

for(k2=0;k2¡MATSIZE+1;++k2){
for(k1=0;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=0;
}
}
for(k2=0;k2¡MATSIZE;++k2){
rho=sqrt(W[mat][k2][k2]*W[mat][k2][k2]+
W[mat][MATSIZE][k2]*W[mat][MATSIZE][k2]);
if(rho == 0){/*if(mat==0){printf("below\n");
}*/goto below4;}
sigma=W[mat][k2][k2]/rho;
tau=W[mat][MATSIZE][k2]/rho;
for(k1=k2;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=W[mat][k2][k1]*sigma+W[mat][MATSIZE][k1]*tau;
Wp[MATSIZE][k1]=-W[mat][k2][k1]*tau+W[mat][MATSIZE][k1]*sigma;
}
Wp[MATSIZE][k2]=0;
for(k1=0;k1¡MATSIZE+1;++k1){
W[mat][k2][k1]=Wp[k2][k1];
W[mat][MATSIZE][k1]=Wp[MATSIZE][k1];
}
below4:;
}
}
```

The following subroutine re-initializes the W matrices.

```
reinitw()
{
int n1,n2,n3;
for(n1=0;n1¡7;++n1){
for(n2=0;n2¡30;++n2){
for(n3=0;n3¡30;++n3){
```

```
W[n1][n2][n3]=0;
}
}
}
}
```

The following subroutine uses the W matrices to calculate
the updated weights.

```
weightch(level,MATSIZE,mat)
int level,MATSIZE,mat;
{
double wchangep[30],a[30],zzt,zt,wp[30],den,fac;
int k,i,zer,n1,n2,n3,counts;

zt=0;
zer=0;
if(level==2){
for(n1=0;n1¡MATSIZE;++n1){
wp[n1]=w[level][mat-1][n1];
}
}
if(level==1){
counts=0;
for(n1=0;n1¡N[Nlevels-1];++n1){
for(n2=0;n2¡N[Nlevels-2]+1;++n2){
wp[counts]=w[level][n1][n2];
++counts;
}
}
}
for(n1=0;n1¡MATSIZE;++n1){
if(fabs(W[mat][n1][n1])¡1e-28){zer=1;}
}
if(zer==0){
a[MATSIZE-1]=W[mat][MATSIZE-1][MATSIZE]/W[mat][MATSIZE-1][MATSIZE-1];
for(n1=MATSIZE-2;n1¿=0;-n1){
a[n1]=W[mat][n1][MATSIZE];
for(n2=MATSIZE-1;n2¿n1;-n2){
a[n1]-=W[mat][n1][n2]*a[n2];
}
a[n1]/=W[mat][n1][n1];
}
}
else{
if(level==2){
```

```
printf("no weight update level 2\n");
for(n1=0;n1¡MATSIZE;++n1){
a[n1]=wp[n1];
}
}
if(level==1){
printf("no weight update level 1\n");
for(n1=0;n1¡MATSIZE;++n1){
a[n1]=wp[n1];
}
}
}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]=a[n1]-wp[n1];
}
den=0;
for(n1=0;n1¡MATSIZE;++n1){
den+=wchangep[n1]*wchangep[n1];
}
den=sqrt(den);
if( den ¿ wchange){fac=wchange/den;}
else{fac=1;}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]*=fac;
}
for(n1=0;n1¡MATSIZE;++n1){
wa[n1]=wp[n1]+wchangep[n1];
}
for(n1=0,zt=0;n1¡MATSIZE;++n1){
if(wa[n1]¿700){wa[n1]=700;zt=1;}
if(wa[n1]¡-700){wa[n1]=-700;zt=1;}
}

if(zt==1){printf("weight at 700\n");}
}
```

The following subroutine implements the forgetting factor by multiplyint the W matrices by the constant ASweight.

```
ASw()
{
int n1,n2,n3;

for(n1=0;n1¡7;++n1){
for(n2=0;n2¡30;++n2){
for(n3=0;n3¡30;++n3){
```

```
W[n1][n2][n3]*=ASweight;
}
}
}
}
```

Network-wise Algorithm

```
#include¡stdio.h¿
#include¡math.h¿
#include¡stdlib.h¿

#define Nlevels 2
/*
This program simulates a N0-input, N2-output 2-layer, feedforward
neural net using the Azimi-Sadjadi algorithm with G-R and
includes biasing. network wise updating.
*/

int datapts,xdata,ydata,numweight,zed,use,Nloops,N[3];
float x[4][519],t[4][519];
float w[3][10][10],wa[30],wpp[10][10];
float K[3][10],b[3][10],delta1,delta2;
float W[1][50][50];
float y[3][10],wchange,err1;
float z[50],tp;
float ASweight;
double err/*,E[1000]*/;
char outfile[40];
FILE *ifp1,*ofp1,*ofp2,*ofp3;

main()
{
int k1,k2,k3;
void AS2();


printf("Enter name for input file: ");
gets(outfile);
if((ifp1=fopen(outfile,"r"))==NULL)
{
printf("fopen failed\n");
exit(0);
}


/*printf("Enter name for output file1: ");
gets(outfile);
if((ofp1=fopen(outfile,"w"))==NULL)
```

```
{
printf("fopen failed\n");
exit(0);
}*/


printf("Enter number of hidden nodes: ");
scanf("%d",&N[1]);

Here the input training set is read.
fscanf(ifp1,"%d,%d,%d\n",&datapts,&xdata,&ydata);
N[0]=xdata;
N[2]=ydata;
printf("datapts=%d xdata=%d ydata=%d\n",
datapts,xdata,ydata);
for(k1=0;k1¡datapts;++k1){
for(k2=0;k2¡xdata;++k2){
fscanf(ifp1,"%f,",&x[k2][k1]);
}
for(k2=0;k2¡ydata;++k2){
fscanf(ifp1,"%f,",&t[k2][k1]);
}
fscanf(ifp1,"\n");
}

printf("Enter maximum number of loops: ");
scanf("%d",&Nloops);
ASweight=0.1;
y[1][N[1]]=1;
y[0][N[0]]=1;
for(k1=0;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1]+1;++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
w[k1][k2][k3]=0.001;
}
}
}

reinitw();
wchange=.4;

AS2();

fprintf(ofp1,"%d,%d,%d\n",N[0],N[1],N[2]);
```

```
/*for(k1=1;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1];++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
fprintf(ofp1,"%f\n",w[k1][k2][k3]);
printf("w[%d][%d][%d]=%f\n",
k1,k2,k3,w[k1][k2][k3]);
}
}
}*/

fclose(ifp1);
fclose(ofp1);
}

void AS2()
{
int n1,n2,n3,count,loop,data,test;

for(loop=0;loop¡Nloops;++loop){
zed=0;
err=0;
use=0;
ASw();
for(data=0;data¡datapts;++data){
for(n1=0;n1¡N[0];++n1){
y[0][n1]=x[n1][data];
}
Below the outputs of the nodes and the derivatives are calculated.
The sum of square errors is calculated below.
for(n1=1;n1¡Nlevels+1;++n1){
for(n2=0;n2¡N[n1];++n2){
y[n1][n2]=0;
for(n3=0;n3¡N[n1-1]+1;++n3){
y[n1][n2]+=y[n1-1][n3]*w[n1][n2][n3];
}
if(y[n1][n2]¡-20){y[n1][n2]=-20;}
b[n1][n2]=y[n1][n2];
y[n1][n2]=1/(1+exp(-y[n1][n2]));
K[n1][n2]=y[n1][n2]*(1-y[n1][n2]);
b[n1][n2]=y[n1][n2]-K[n1][n2]*b[n1][n2];
}
}
count=0;
Here the linearized intputs and outputs are calculated.
tp=t[0][data]-b[Nlevels][0]
```

```
for(n2=0;n2¡N[Nlevels-1];++n2){
z[count]=K[Nlevels][0]*b[Nlevels-1][n2];
++count;
}
z[count]=K[Nlevels][0]*y[Nlevels-1][N[Nlevels-1]];
++count;
for(n2=0;n2¡N[Nlevels-1];++n2){
for(n3=0;n3¡N[Nlevels-2]+1;++n3){
z[count]=K[Nlevels][0]*K[Nlevels-1][n2]*y[Nlevels-2][n3];
++count;
}
}
rotin(N[Nlevels-1]+1+(N[Nlevels-1]*(N[Nlevels-2]+1)),0);

for(n1=0,err1=0,test=0;n1¡ydata;++n1){
err1+=(t[n1][data]-y[2][n1])*(t[n1][data]-y[2][n1]);
if(fabs(t[n1][data]-y[2][n1]) ¿= 0.5){test=1;}
}
if(test==0){zed++;}
err+=err1;
}
```

Below the new weights are calculated.

```
weightch(N[Nlevels-1]+1+(N[Nlevels-1]*(N[Nlevels-2]+1)),0);
count=0;
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
w[Nlevels][0][n2]=wa[count];
++count;
}
for(n1=0;n1¡N[Nlevels-1];++n1){
for(n2=0;n2¡N[Nlevels-2]+1;++n2){
if(fabs(w[Nlevels][0][n1])¿1e-10){
wpp[n1][n2]=wa[count];
w[Nlevels-1][n1][n2]=wpp[n1][n2]/w[Nlevels][0][n1];
}
++count;
}
}
printf("loop=%d data=%d zed=%d err[data]=%f\n",
loop,data,zed,err);

}
}
```

The following subroutine rotates the linearized training patterns into the W matrices using QR decomposition.

```
rotin(MATSIZE,mat)
int MATSIZE,mat;
{
int k1,k2;
float rho,sigma,tau,Wp[50][50];


for(k2=0;k2¡MATSIZE;++k2){
W[mat][MATSIZE][k2]=z[k2];
}
W[mat][MATSIZE][MATSIZE]=tp;

for(k2=0;k2¡MATSIZE+1;++k2){
for(k1=0;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=0;
}
}
for(k2=0;k2¡MATSIZE;++k2){
rho=sqrt(W[mat][k2][k2]*W[mat][k2][k2]+
W[mat][MATSIZE][k2]*W[mat][MATSIZE][k2]);
if(rho == 0){/*if(mat==0){printf("below\n");
}*/goto below4;}
sigma=W[mat][k2][k2]/rho;
tau=W[mat][MATSIZE][k2]/rho;
for(k1=k2;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=W[mat][k2][k1]*sigma+W[mat][MATSIZE][k1]*tau;
Wp[MATSIZE][k1]=-W[mat][k2][k1]*tau+W[mat][MATSIZE][k1]*sigma;
}
Wp[MATSIZE][k2]=0;
for(k1=0;k1¡MATSIZE+1;++k1){
W[mat][k2][k1]=Wp[k2][k1];
W[mat][MATSIZE][k1]=Wp[MATSIZE][k1];
}
below4:;
}

}
```

The following subroutine re-initializes the W matrices.

```
reinitw()
{
int n1,n2,n3;
for(n1=0;n1¡1;++n1){
for(n2=0;n2¡50;++n2){
for(n3=0;n3¡50;++n3){
```

```
W[n1][n2][n3]=0;
}
}
}

}
```

The following subroutine uses the W matrices to calculate the
updated weights.

```
weightch(MATSIZE,mat)
int MATSIZE,mat;
{
double wchangep[50],a[50],zzt,zt,wp[50],den,fac;
int k,i,n1,n2,n3,counts;

zt=0;

counts=0;
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
wp[counts]=w[Nlevels][0][n2];
++counts;
}
for(n1=0;n1¡N[Nlevels-1];++n1){
for(n2=0;n2¡N[Nlevels-2]+1;++n2){
wp[counts]=wpp[n1][n2];
++counts;
}
}

if(fabs(W[mat][MATSIZE-1][MATSIZE-1])¿1e-28){
a[MATSIZE-1]=W[mat][MATSIZE-1][MATSIZE]/W[mat][MATSIZE-1][MATSIZE-1];
}
else{
printf("no weight update %d\n",MATSIZE);
a[MATSIZE-1]=wp[MATSIZE-1];
}
for(n1=MATSIZE-2;n1¿=0;-n1){
if(fabs(W[mat][n1][n1])¿1e-28){
a[n1]=W[mat][n1][MATSIZE];
for(n2=MATSIZE-1;n2¿n1;-n2){
a[n1]-=W[mat][n1][n2]*a[n2];

}
a[n1]/=W[mat][n1][n1];
}
```

```
else{
printf("no weight update %d\n",MATSIZE);
a[MATSIZE-1]=wp[MATSIZE-1];
}
}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]=a[n1]-wp[n1];
}
den=0;
for(n1=0;n1¡MATSIZE;++n1){
den+=wchangep[n1]*wchangep[n1];
}
den=sqrt(den);
if( den ¿ wchange){fac=wchange/den;}
else{fac=1;}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]*=fac;
}
for(n1=0;n1¡MATSIZE;++n1){
wa[n1]=wp[n1]+wchangep[n1];
}
for(n1=0,zt=0;n1¡MATSIZE;++n1){
if(wa[n1]¿700){wa[n1]=700;zt=1;}
if(wa[n1]¡-700){wa[n1]=-700;zt=1;}
}

if(zt==1){printf("weight at 700\n");}
}
```

The following subroutine implements the forgetting factor by multiplyint the W matrices by the constant ASweight.

```
ASw()
{
int n1,n2,n3;

for(n1=0;n1¡1;++n1){
for(n2=0;n2¡50;++n2){
for(n3=0;n3¡50;++n3){
W[n1][n2][n3]*=ASweight;
}
}
}
}
```

Layer-wise algorithm with data reduction incorporated

```
#include¡stdio.h¿
#include¡math.h¿
#include¡stdlib.h¿

/*
This program simulates a N0-input, N2-output 2-layer, feedforward
neural net using the Azimi-Sadjadi algorithm with G-R and
includes biasing. Includes data reduction.
*/

int datapts,xdata,ydata,numweight,zed,use[6],Nloops,N[3],Nlevels;
float x[2][519],t[4][519],dwest[5][519],delta1,delta2;
float w[3][10][10],wa[30],b[3][10],K[3][10],W[5][30][30],Wpp[30];
float y[3][10],wchange,err1,z[30],tp,ASweight;
float wnorm[5],bnorm[5],wb[5][30],olderr;
double err,wow[6];
char outfile[40];
FILE *ifp1,*ofp1;

main()
{
int k1,k2,k3;
void AS2();


printf("Enter name for input file: ");
gets(outfile);
if((ifp1=fopen(outfile,"r"))==NULL)
{
printf("fopen failed\n");
exit(0);
}


/*printf("Enter name for output file1: ");
gets(outfile);
if((ofp1=fopen(outfile,"w"))==NULL)
{
printf("fopen failed\n");
```

```
exit(0);
}*/

printf("Enter number of hidden nodes: ");
scanf("%d",&N[1]);

fscanf(ifp1,"%d,%d,%d\n",&datapts,&xdata,&ydata);
N[0]=xdata;
N[2]=ydata;
printf("datapts=%d xdata=%d ydata=%d\n",
datapts,xdata,ydata);
for(k1=0;k1¡datapts;++k1){
for(k2=0;k2¡xdata;++k2){
fscanf(ifp1,"%f,",&x[k2][k1]);
}
for(k2=0;k2¡ydata;++k2){
fscanf(ifp1,"%f,",&t[k2][k1]);
}
fscanf(ifp1,"\n");
}

printf("Enter maximum number of loops: ");
scanf("%d",&Nloops);
Nlevels=2;
ASweight=0.1;
olderr=0;
y[1][N[1]]=1;
y[0][N[0]]=1;
for(k1=0;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1]+1;++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
w[k1][k2][k3]=0.001;
}
}
}

for(k1=0;k1¡5;++k1){wow[k1]=0;}
reinitw();
wchange=.4;

AS2();
for(k1=0;k1¡6;++k1){
wow[k1]/=(Nloops-10);
printf("wow[%d]=%f ",k1,wow[k1]);
/* fprintf(ofp1,"wow[%d]=%f ",k1,wow[k1]);*/
```

```
}
printf("\n");
/* fprintf(ofp1,"\n");*/


/*fprintf(ofp1,"%d,%d,%d\n",N[0],N[1],N[2]);
for(k1=1;k1¡Nlevels+1;++k1){
for(k2=0;k2¡N[k1];++k2){
for(k3=0;k3¡N[k1-1]+1;++k3){
fprintf(ofp1,"%f\n",w[k1][k2][k3]);
printf("w[%d][%d][%d]=%f\n",
k1,k2,k3,w[k1][k2][k3]);
}
}
}*/

fclose(ifp1);
fclose(ofp1);
}

void AS2()
{
int n1,n2,n3,count,loop,data,test;
float mean;

for(n2=0;n2¡5;++n2){for(n1=0;n1¡datapts;++n1)
{dwest[n2][n1]=1;}}

for(loop=0;loop¡Nloops;++loop){
zed=0;
err=0;
for(n1=0;n1¡5;++n1){use[n1]=0;}
The data reduction starts after the 10^th iteration.
if(loop¿9){
norms();
for(data=0;data¡datapts;++data){
for(n1=0;n1¡N[0];++n1){
y[0][n1]=x[n1][data];
}
for(n1=1;n1¡Nlevels+1;++n1){
for(n2=0;n2¡N[n1];++n2){
y[n1][n2]=0;
for(n3=0;n3¡N[n1-1]+1;++n3){
y[n1][n2]+=y[n1-1][n3]*w[n1][n2][n3];
}
```

```
if(y[n1][n2]¡-20){y[n1][n2]=-20;}
b[n1][n2]=y[n1][n2];
y[n1][n2]=1/(1+exp(-y[n1][n2]));
K[n1][n2]=y[n1][n2]*(1-y[n1][n2]);
b[n1][n2]=y[n1][n2]-K[n1][n2]*b[n1][n2];
}
}
for(n1=0;n1¡N[Nlevels];++n1){
tp=t[n1][data]-b[Nlevels][n1];
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
z[n2]=K[Nlevels][n1]*y[Nlevels-1][n2];
}
reinitwpp();
rotin1(N[Nlevels-1]+1,n1+1);
dbdA(N[Nlevels-1]+1,n1+1,data);
}
reinitwpp();
for(n1=0;n1¡N[Nlevels];++n1){
count=0;
tp=t[n1][data]-b[Nlevels][n1];
for(n2=0;n2¡N[Nlevels-1];++n2){
tp-=K[Nlevels][n1]*w[Nlevels][n1][n2]*b[Nlevels-1][n2];
for(n3=0;n3¡N[Nlevels-2]+1;++n3){
z[count]=K[Nlevels][n1]*K[Nlevels-1][n2]*
w[Nlevels][n1][n2]*y[Nlevels-2][n3];
++count;
}
}
tp-=K[Nlevels][n1]*w[Nlevels][n1][n2]*y[Nlevels-1][n2];
rotin1(N[Nlevels-1]*(N[Nlevels-2]+1),0);
}
dbdA(N[Nlevels-1]*(N[Nlevels-2]+1),0,data);
}
for(n1=0;n1¡N[Nlevels]+1;++n1){
mean=0;
for(n2=0;n2¡datapts;++n2){
if(mean¡dwest[n1][n2]){mean=dwest[n1][n2];}
}
mean/=(2);
for(n2=0;n2¡datapts;++n2){
if(dwest[n1][n2]¿mean —— dwest[n1][n2]¡0){dwest[n1][n2]=1;}
else{dwest[n1][n2]=0;}
}
}
}
```

```
ASw();
for(data=0;data¡datapts;++data){
for(n1=0;n1¡N[0];++n1){
y[0][n1]=x[n1][data];
}
for(n1=1;n1¡Nlevels+1;++n1){
for(n2=0;n2¡N[n1];++n2){
y[n1][n2]=0;
for(n3=0;n3¡N[n1-1]+1;++n3){
y[n1][n2]+=y[n1-1][n3]*w[n1][n2][n3];
}
if(y[n1][n2]¡-20){y[n1][n2]=-20;}
b[n1][n2]=y[n1][n2];
y[n1][n2]=1/(1+exp(-y[n1][n2]));
K[n1][n2]=y[n1][n2]*(1-y[n1][n2]);
b[n1][n2]=y[n1][n2]-K[n1][n2]*b[n1][n2];
}
}
for(n1=0;n1¡N[Nlevels];++n1){
if(dwest[n1+1][data]¿0){
tp=t[n1][data]-b[Nlevels][n1];
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
z[n2]=K[Nlevels][n1]*y[Nlevels-1][n2];
}
rotin(N[Nlevels-1]+1,n1+1);
++use[n1+1];
}
}
if(dwest[0][data]¿0){
for(n1=0;n1¡N[Nlevels];++n1){
count=0;
tp=t[n1][data]-b[Nlevels][n1];
for(n2=0;n2¡N[Nlevels-1];++n2){
tp-=K[Nlevels][n1]*w[Nlevels][n1][n2]*b[Nlevels-1][n2];
for(n3=0;n3¡N[Nlevels-2]+1;++n3){
z[count]=K[Nlevels][n1]*K[Nlevels-1][n2]*
w[Nlevels][n1][n2]*y[Nlevels-2][n3];
++count;
}
}
tp-=K[Nlevels][n1]*w[Nlevels][n1][n2]*y[Nlevels-1][n2];
rotin(N[Nlevels-1]*(N[Nlevels-2]+1),0);
}
++use[0];
}
```

```
for(n1=0,err1=0,test=0;n1¡ydata;++n1){
err1+=(t[n1][data]-y[2][n1])*(t[n1][data]-y[2][n1]);
if(fabs(t[n1][data]-y[2][n1]) ¿= 0.5){test=1;}
}
if(test==0){zed++;}
err+=err1;
}

for(n1=0;n1¡N[Nlevels];++n1){
weightch(Nlevels,N[Nlevels-1]+1,n1+1);
for(n2=0;n2¡N[Nlevels-1]+1;++n2){
w[Nlevels][n1][n2]=wa[n2];
}
}
weightch(Nlevels-1,N[Nlevels-1]*(N[Nlevels-2]+1),0);
count=0;
for(n1=0;n1¡N[Nlevels-1];++n1){
for(n2=0;n2¡N[Nlevels-2]+1;++n2){
w[Nlevels-1][n1][n2]=wa[count];
++count;
}
}
printf("loop=%d data=%d zed=%d err[data]=%f\n",
loop,data,zed,err);
fprintf(ofp1,"%d,%d,%f",loop,zed,err);
for(n1=0;n1¡5;++n1){
printf("use[%d]=%d ",n1,use[n1]);
if(loop¿9){
wow[n1]+=use[n1];
}
}
printf("\n");
}
}


rotin(MATSIZE,mat)
int MATSIZE,mat;
{
int k1,k2;
float rho,sigma,tau,Wp[30][30];

for(k2=0;k2¡MATSIZE;++k2){
W[mat][MATSIZE][k2]=z[k2];
}
```

```
W[mat][MATSIZE][MATSIZE]=tp;

for(k2=0;k2¡MATSIZE+1;++k2){
for(k1=0;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=0;
}
}
for(k2=0;k2¡MATSIZE;++k2){
rho=sqrt(W[mat][k2][k2]*W[mat][k2][k2]+
W[mat][MATSIZE][k2]*W[mat][MATSIZE][k2]);
if(rho == 0){/*if(mat==0){printf("below\n");
}*/goto below4;}
sigma=W[mat][k2][k2]/rho;
tau=W[mat][MATSIZE][k2]/rho;
for(k1=k2;k1¡MATSIZE+1;++k1){
Wp[k2][k1]=W[mat][k2][k1]*sigma+W[mat][MATSIZE][k1]*tau;
Wp[MATSIZE][k1]=-W[mat][k2][k1]*tau+W[mat][MATSIZE][k1]*sigma;
}
Wp[MATSIZE][k2]=0;
for(k1=0;k1¡MATSIZE+1;++k1){
W[mat][k2][k1]=Wp[k2][k1];
W[mat][MATSIZE][k1]=Wp[MATSIZE][k1];
}
below4:;
}
}

rotin1(MATSIZE,mat)
int MATSIZE,mat;
{
int k1,k2;
float rho,sigma,tau,Wp[30];

for(k2=0;k2¡MATSIZE;++k2){
W[mat][MATSIZE][k2]=z[k2];
}
W[mat][MATSIZE][MATSIZE]=tp;

for(k1=0;k1¡MATSIZE+1;++k1){
Wp[k1]=0;
}
rho=sqrt(W[mat][0][0]*W[mat][0][0]+
W[mat][MATSIZE][0]*W[mat][MATSIZE][0]);
if(rho == 0){goto below4;}
sigma=W[mat][0][0]/rho;
```

```
tau=W[mat][MATSIZE][0]/rho;
for(k1=0;k1¡MATSIZE+1;++k1){
Wpp[k1]=W[mat][0][k1]*sigma+W[mat][MATSIZE][k1]*tau;
}
below4:;
}
```

This subroutine calculates approximation to the weight change vector.

```
dbdA(MATSIZE,mat,datas)
int MATSIZE,mat,datas;
{
int k1;
float dbdAnorm,dAnorm;

dbdAnorm=0;
dAnorm=0;
for(k1=0;k1¡MATSIZE;++k1){
dbdAnorm+=(W[mat][0][k1]-Wpp[k1])*wb[mat][k1];
dAnorm+=fabs(W[mat][0][k1]-Wpp[k1]);
}
dbdAnorm=fabs((W[mat][0][MATSIZE]-Wpp[MATSIZE])-dbdAnorm);
dwest[mat][datas]=(dbdAnorm)/(bnorm[mat]-wnorm[mat]*dAnorm);
}


weightch(level,MATSIZE,mat)
int level,MATSIZE,mat;
{
double wchangep[30],a[30],zzt,zt,wp[30],den,fac;
int k,i,zer,n1,n2,n3,counts;

zt=0;
zer=0;
if(level==2){
for(n1=0;n1¡MATSIZE;++n1){
wp[n1]=w[level][mat-1][n1];
}
}
if(level==1){
counts=0;
for(n1=0;n1¡N[Nlevels-1];++n1){
for(n2=0;n2¡N[Nlevels-2]+1;++n2){
wp[counts]=w[level][n1][n2];
++counts;
}
```

```
}
}
for(n1=0;n1¡MATSIZE;++n1){
if(fabs(W[mat][n1][n1])¡1e-28){zer=1;}
}
if(zer==0){
a[MATSIZE-1]=W[mat][MATSIZE-1][MATSIZE]/W[mat][MATSIZE-1][MATSIZE-1];
for(n1=MATSIZE-2;n1¿=0;-n1){
a[n1]=W[mat][n1][MATSIZE];
for(n2=MATSIZE-1;n2¿n1;-n2){
a[n1]-=W[mat][n1][n2]*a[n2];
}
a[n1]/=W[mat][n1][n1];
}
}
else{
if(level==2){
printf("no weight update level 2\n");
for(n1=0;n1¡MATSIZE;++n1){
a[n1]=wp[n1];
}
}
if(level==1){
printf("no weight update level 1\n");
for(n1=0;n1¡MATSIZE;++n1){
a[n1]=wp[n1];
}
}
}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]=a[n1]-wp[n1];
}
den=0;
for(n1=0;n1¡MATSIZE;++n1){
den+=wchangep[n1]*wchangep[n1];
}
den=sqrt(den);
if( den ¿ wchange){fac=wchange/den;}
else{fac=1;}
for(n1=0;n1¡MATSIZE;++n1){
wchangep[n1]*=fac;
}
for(n1=0;n1¡MATSIZE;++n1){
wa[n1]=wp[n1]+wchangep[n1];
}
```

```
for(n1=0,zt=0;n1¡MATSIZE;++n1){
if(wa[n1]¿700){wa[n1]=700;zt=1;}
if(wa[n1]¡-700){wa[n1]=-700;zt=1;}
}

if(zt==1){printf("weight at 700\n");}
}

reinitw()
{
int n1,n2,n3;
for(n1=0;n1¡5;++n1){
for(n2=0;n2¡30;++n2){
for(n3=0;n3¡30;++n3){
W[n1][n2][n3]=0;
}
}
}
}
```

The subroutine below initializes the Wpp vector to zero.

```
reinitwpp()
{
int n1,n2,n3;
for(n1=0;n1¡30;++n1){
Wpp[n1]=0;
}
}
```

This subroutine calculates the norms for the weight vectors and the b vector.

```
norms()
{
int k1,k2,k3;

for(k1=0;k1¡N[2]+1;++k1){
wnorm[k1]=0;
bnorm[k1]=0;
}
for(k1=1;k1¡N[2]+1;++k1){
for(k2=0;k2¡N[1]+1;++k2){
if(wnorm[k1]¡fabs(wb[k1][k2])){wnorm[k1]=fabs(wb[k1][k2]);}
}
}
for(k2=0;k2¡N[0]+1;++k2){
```

```
if(wnorm[0]¡fabs(wb[0][k2])){wnorm[0]=fabs(wb[0][k2]);}
}
for(k1=1;k1¡N[2]+1;++k1){
for(k2=0;k2¡N[1]+1;++k2){
if(bnorm[k1]¡fabs(W[k1][k2][N[1]+1]))
{bnorm[k1]=fabs(W[k1][k2][N[1]+1]);}
}
}
for(k2=0;k2¡N[0]+1;++k2){
if(bnorm[0]¡fabs(W[0][k2][N[1]*(N[0]+1)]))
{bnorm[0]=fabs(W[0][k2][N[1]*(N[0]+1)]);}
}
}

ASw()
{
int k1,k2,k3;
for(k1=0;k1¡N[2]+1;++k1){
for(k2=0;k2¡30;++k2){
for(k3=0;k3¡30;++k3){
W[k1][k2][k3]*=ASweight;
}
}
}
}
```