12-1991

A comparative evaluation of heuristics used to improve convergence rates of the back-propagation algorithm

Kenneth Sherman Noggle

To the Graduate Council:

I am submitting herewith a thesis written by Kenneth Sherman Noggle entitled "A comparative evaluation of heuristics used to improve convergence rates of the back-propagation algorithm." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Reinhold Mann, Major Professor

We have read this thesis and recommend its acceptance:

Michael Thomason, Bruce MacLennan

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Kenneth Sherman Noggle entitled *A Comparative Evaluation of Heuristics used to Improve Convergence Rates of the Back-Propagation Algorithm.* I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Reinhold Mann

Reinhold Mann, Major Professor

We have read this thesis and recommend its acceptance:

Bruce MacLennan

Michael G Thomason

Accepted for the Council:

CW Minkel

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Kenneth S. Vogle

Date 15 Aug 91

A COMPARATIVE EVALUATION OF HEURISTICS USED TO IMPROVE CONVERGENCE RATES OF THE BACK-PROPAGATION ALGORITHM

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Kenneth Sherman Noggle
December 1991

DEDICATION

This thesis is dedicated to the memory of my father
Dr. Thomas Sherman Noggle
whose ingenuity still inspires,
and to my wonderful daughter
Emily Victoria,
if only they could have met.

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Reinhold Mann, for his advice, time, and patience throughout the course of this thesis project. I would also like to thank my committee members, Dr. Michael Thomason, and Dr. Bruce MacLennan for their valuable input. I would like to express my thanks, love, and appreciation to my wife, Joanne Filchock, for her patience and understanding throughout the duration. I would also like to thank my mother, Alice Noggle, for her unending generosity. Their contributions were vital to the successful completion of this work.

Abstract

Artificial neural networks exhibit important classification capabilities in a variety of pattern recognition applications, due in large part to their ability to adapt to a changing environment without requiring either a complex set of programmed steps or underlying sample distribution information. Their adaptive learning capability may allow them to be successfully applied to problems in such areas as speech and pattern recognition [Lippmann 87]. Multi-layer feed-forward neural networks classifiers require a training rule for adjusting weights in internal layers. One such training rule, the back-propagation of error algorithm, also known as the generalized delta rule [Rumelhart 86a], has become one of the most widely used training algorithms for neural networks. However, it suffers from two major drawbacks: slow convergence rates and convergence to non-optimal solutions, or local minima.

The objective of this research was to investigate the effect different heuristics have on the performance of the standard back-propagation (BP) algorithm. The methods that were studied included modification of both learning rate and momentum parameters, several adaptive techniques (e.g. the Delta-Bar-Delta and Extended Delta-Bar-Delta algorithms) for dynamically adjusting these parameters, modification of the sigmoid function, and replacement of the random initial weights with values which pre-partition the inputs into separate decision regions. Results from the use of these heuristics were compared in terms of both their overall impact on convergence rates and their effect on convergence to local minima. These heuristics were implemented and tested on three benchmark problems, each with different characteristics which presented varying levels of difficulty to the neural network. The three problem types used to test and compare the algorithms were the XOR (parity), multiplexer, and encoder/decoder problems. After benchmarking performance on small problem sets, problem size was increased to examine the effect of scaling on the different heuristics.

The adaptive algorithms were shown to achieve reduced convergence times compared to BP on all problem types and sizes. The improvement was more pronounced for the larger problems. The Delta-Bar-Delta algorithm often produced the best results of any of the heuristics using only three additional inputs. The large number of parameters required by the Extended Delta-Bar-Delta algorithm made it difficult to tune. In order to converge to solutions on scaled-up problem sizes, the BP algorithm was shown to require normalization of the weight update value, as well as an individual per-pattern error criterion in place of the sum of squared error criterion. Computational requirements were shown to increase exponentially with additional input lines for the XOR/Parity and multiplexer problems, and polynomially for the encoder/decoder problem.

Contents

List of Tables

List of Figures

Symbol Table

Back-Propagation Algorithm

α back-propagation algorithm momentum coefficient.

η back-propagation algorithm learning rate coefficient.

i, j, k indices for nodes in input, hidden and output layers, respectively.

p training set pattern index.

t epoch (complete pass through training set) index.

θ_j, θ_k bias term (threshold) for node j (k) in hidden layer (output) layer.

net_{pj} net sum of product of inputs and weights at node j (in hidden layer) as a result of the application of pattern p.

o_{pj}, o_{pk} output activation of node j in hidden layer (node k in output layer) as a result of the application of pattern p from the training set.

δ_{pj}, δ_{pk} error signal for node j in hidden layer (node k in output layer) as a result of the application of pattern p from the training set.

$w_{ij}(t+1)$ connection weight from node i in input layer to node j in hidden layer after update resulting from presentation of t^{th} training set.

$w(t)$ generic connection weight at t^{th} epoch.

$\Delta_p w_{ij}, \Delta_p w_{jk}$ change in weight w_{ij}, (w_{jk}) resulting from application of pattern p from training set.

$\Delta w_{ij}, \Delta w_{jk}$ change in weight w_{ij}, (w_{jk}) resulting from application of entire training set.

x_{pi} i^{th} component of input vector (pattern p at input node i).

t_{pk}	k^{th} component of target vector for pattern p.
E_{pk}	error at node k in output layer resulting from pattern p.
E_p	sum of squared pattern error.
$\frac{\partial E_p}{\partial w_{ij}}$	partial derivative of the error with respect to weight w_{ij}.
E_{TOT}	total sum of squared error over all output nodes, for all patterns.
f	sigmoid activation function.
$\frac{\partial o_{pj}}{\partial net_{pj}}$	partial derivative of the sigmoid function.

Gradient Reuse Algorithm

$\Delta_p g_{ij}$	error gradient with weight w_{ij} resulting from pattern p.
g_{ij}	error gradient with weight w_{ij} resulting from training set.
μ	adaptable convergence parameter (η in BP).

Delta Bar Delta Algorithm

$\epsilon_{ij}(t)$	learning rate associated with weight $w_{ij}(t)$.
$\Delta\epsilon(t)$	change in ϵ at epoch t.
$\frac{\partial J(t)}{\partial w(t)}$	partial derivative of the error with respect to weight $w(t)$.
$\bar{\delta}(t)$	exponential average of current and past error derivatives.
$\eta_{ij}(t)$	weight specific learning rate coefficient.
κ	learning rate update constant.
ϕ	learning rate reduction proportion.
θ	$\bar{\delta}$ update coefficient.

Extended Delta Bar Delta Algorithm

$\alpha_{ij}(t)$	weight specific adaptive momentum coefficient.
η_{max}	maximum value for learning rate coefficient.
α_{max}	maximum value for momentum coefficient.
κ_l	learning rate update coefficient.
γ_l	learning rate increase coefficient for *exp* function.
ϕ_l	learning rate reduction proportion.
κ_m	momentum update coefficient.
γ_m	momentum increase coefficient for *exp* function.
ϕ_m	momentum reduction proportion.
λ	error tolerance parameter.

Chapter 1

Introduction

1.1 Artificial Neural Networks

Artificial neural networks may be viewed as mathematical models consisting of multiple nonlinear processing elements (called nodes or neurons) interconnected with weighted links which are adaptively modified during training in order to produce desired outputs for the given input data. The neurons are typically arranged in multiple layers. The input layer serves to fan out inputs to all nodes in the next layer. Nodes in each layer operate in parallel on the output of the previous layer. Individual neurons perform a simple operation, consisting of the application of a nonlinear function to a weighted sum of its inputs. The difference between the response of the network to an input and the corresponding desired output is used to effect adjustments in the weights between neighboring nodes. An internal representation of the externally applied inputs is thus contained in the strength of the connections between neurons.

Artificial neurons may be organized into architectures ranging from single nodes up to large, multi-layer networks without requiring additional explicit training instructions due to the change in size. The ability to scale up the size of networks to handle larger problems without having to create a new training procedure for each added node may be found to be beneficial as networks begin to operate on larger and much more complex problems, such as those found in speech recognition. The redundancy in information storage created by the high interconnectedness of nodes may also produce a degree of robustness or fault tolerance not usually associated with traditional sequential machine processing. This is an important feature for VLSI implementations of neural networks: the network could still function even after the loss of individual nodes or connections.

Neural networks can successfully emulate existing classification and clustering algorithms, while requiring fewer assumptions about the underlying distributions [Huang 87]. In the BP algorithm, decision boundaries are adjusted to orientations which minimize error. The internal weights, which determine the decision boundaries, are modified iteratively based upon the level of classification error, calculated as the difference between actual and desired output. The network may also be viewed as a mapping through which points in the input (pattern) space are transformed into corresponding points in the output space on the basis of specified attribute values, such as class membership [Pao 89]. Because of their ability to internally generate any arbitrary mapping of inputs to the required outputs [Hecht-Nielsen 90], they may also be better able to deal with non-Gaussian distributions produced by nonlinear processes [Lippmann 87a]. Their ability to emulate the performance of existing classification algorithms is necessary for their application to pattern recognition problems. Equally important, perhaps, is the ability of a network to adapt to change as it learns.

1.2 Neural Network Attributes

A general framework which lists and defines eight major aspects which are part of any neural network model is found in [Rumelhart 86a]. Each network model may be understood and categorized in terms of its particular implementation of each of these aspects. These defining attributes of neural network models are:

- a set of processing units
- a pattern of connectivity
- a state of activation
- an activation rule
- an output function
- a propagation rule
- a learning rule
- an operating environment

The definitions for these attributes [Rumelhart 86a], and their particular form of implementation in the neural networks used, are given below:

Each processing unit receives inputs from neighboring nodes, and as a function of this input, generates an output to its neighbors. Each node generally performs a simple operation, such as producing a weighted sum of its inputs.

The processing units are arranged into a network. The pattern of connectivity (the arrangement of nodes and links), determines how the system will respond to input data.

The state of the network at any time is known as its state of activation. The pattern of activation over all the set of processing units in the network contains the internal representation of the patterns applied to the system.

The output of the processing unit is determined by its state of activation. The strength of its output determines the degree to which it affects its neighbors. Associated with each unit is an output function which maps the current state of activation to its output. The output function may simply be the identity function, or it may be a stochastic function in which the output depends probabilistically upon the activation value, or it may be a threshold function such that a unit with a activation level below a certain value will have no effect on its neighbors.

A propagation rule specifies the mechanism by which the outputs of the units are combined with their respective connections to produce a net input into the next layer of units. The propagation rule typically specifies that the net input to a node is the weighted sum of its inputs.

An activation rule specifies how the net inputs to a unit (specified by the propagation rule) are combined with the current state to produce a new state of activation. This may be the identity function, a threshold function, or a stochastic function. Additionally, it may be necessary that this function be differentiable, in which case, a sigmoid function is often used.

A learning rule specifes the mechanism by which patterns of connectivity are changed based upon the training data. Many learning rules are variants of the Hebbian learning rule in which a connection between two nodes is strengthened with use.

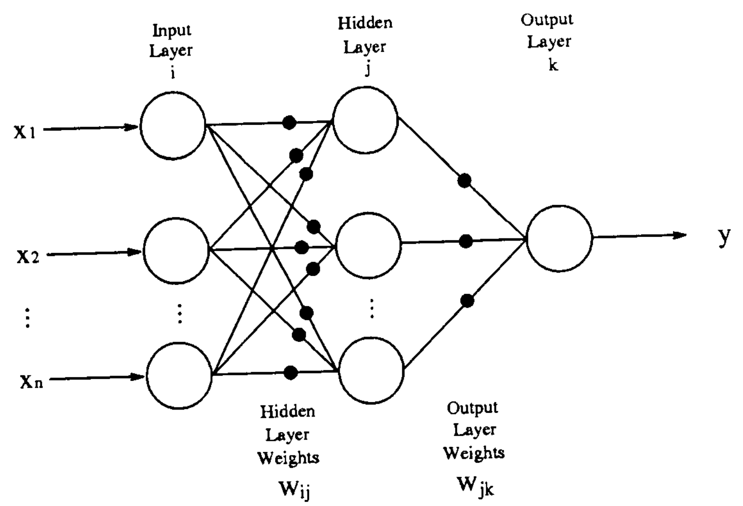
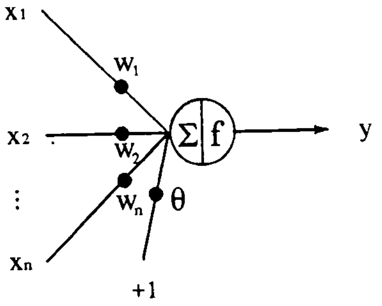


Figure 1.1: Fully-Connected Multi-Layer n-n-1 Artificial Neural Network with Three Layers of Neurons and Two Layers of Trainable Weights.

The operating environment is the set of inputs which are applied to the network. In some models, individual patterns are selected stochastically from the input set. In most cases, however, each pattern in the set of inputs is applied sequentially to the system.

In terms of these attributes, the implementation used in this thesis involves a fully connected, feed-forward network of processing nodes (Figure 1.1), with three layers of neurons and two layers of trainable weights. Inputs to the system are binary, due only to the nature of the problems being studied: analog values could be used. The hidden layer weights, w_{ij}, and the output layer weights, w_{jk}, are shown. The output from each node in the input and hidden layers is used as an input to each node in the subsequent layer. The output from the system is continuously valued. Each processing node produces a single analog output which is the result of applying the nonlinear and differentiable sigmoid function to the weighted sum of its inputs as shown in Figure 1.2. The net activation



$$\text{net} = \sum x_i w_i + \theta$$

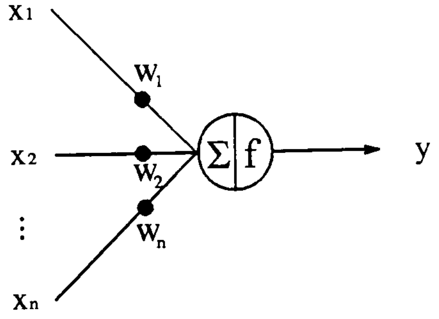
$$y = f(\text{net}) \qquad \text{where: } f(\text{net}) = \frac{1}{1 + \exp^{-\text{net}}}$$

Figure 1.2: An Individual Processing Unit (Neuron) from a Multi-Layer Network with Net Activation Level Calculation and Sigmoid Activation Function.

level is shown as the sum of the weighted inputs plus a trainable threshold term, θ, which is assumed to be connected to a constant input of 1.0. The learning rule consists of the back-propagation algorithm and the heuristic being tested. The operating environment will be the sequential application of training patterns from a training set. These patterns (and the size of the training set) differ based upon the type of problem being processed. The patterns, however, are uniformly distributed in pattern space.

1.3 Neural Network Learning Rules

Current neural network architectures and learning algorithms are similar in some respects to some of the earliest models. The first description of a neuron-like node, in [McCulloch 43], lacked a learning algorithm. This neuron model simply summed the weighted inputs and output a +1 if this sum was not less than a specified threshold value, otherwise a -1 was output.



$$y = f(\Sigma\ x_i w_i - \theta)$$

Figure 1.3: McCulloch & Pitts Neuron Model.

Many learning algorithms are based upon the learning rule proposed in [Hebb 49], in which the strength of a connection between two nodes is increased with repeated use. The perceptron training algorithm explained in [Rosenblatt 62], produced the first trainable neural network. The algorithm stems from ideas found in both the McCulloch & Pitts neuron model and the Hebbian learning rule. Given a network similar to that shown in Figure 1.3, the weight vector representing the weights attached to the input lines leading into the neuron is updated based upon the relationship between the perceptron's actual output and the correct result, as follows:

$$W_{t+1} = W_t + \eta(d - y)X$$

where: X is the input vector, W_t is the weight vector at time t, η is a positive learning rate coefficient ($\eta \leq 1.0$), y is the output of perceptron ($+1$ or -1), and d is the desired target value.

The single layer perceptron model is capable of separating inputs into

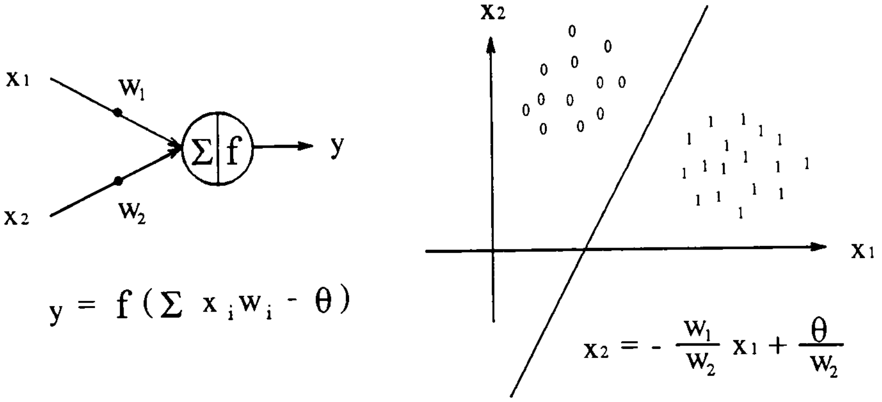


Figure 1.4: Two-Input Perceptron and Resultant Decision Boundary.

two distinct decision regions separated by a hyperplane. A linear boundary separating two linearly separable classes is shown in Figure 1.4. The equation demonstrates how network weights are used to generate a linear discriminant function. Inputs from one side of the boundary produce a positive function value, and are classified together. Inputs producing negative function values are likewise grouped. Error in the classification process is used to adjust the weight values, which changes the location of the decision boundary. Rosenblatt showed that if the two classes are linearly separable, the perceptron convergence procedure is guaranteed to converge in a finite number of iterations.

A similar training algorithm which minimizes the overall classification error with each iteration was described by Widrow and Hoff, in [Widrow 60], as part of the adaptive linear element 'adaline' model. The weight update rule, variously called the delta rule, least mean squared (LMS) rule, or the Widrow-

Hoff update rule modifies weights as follows:

$$W_{t+1} = W_t + \frac{\eta\, \Delta E\, X}{|X|^2}$$

where: ΔE = desired response - actual output

Both the Perceptron and the Widrow-Hoff weight update rules modify the weight vector by a fraction (dependent upon the step size η) of each misclassified pattern vector, X, until all patterns are correctly classifed. Thus, the resultant weight vector is adjusted in a direction which will tend to minimize future misclassifications. In the case of correct classifications, no weight update is made. In the LMS update rule, the weight update is also determined by the magnitude of the error, ΔE, so that relatively small classification errors result in more modest weight adjustments than do larger errors. In Figure 1.5, from [Widrow 87] , the new weight vector, W_{t+1}, is the result of the initial weight vector, W_t, plus ΔW_t, where ΔW_t is parallel to the input vector X_t. Thus, the weight vector changes in the direction of the misclassified vector. This adjustment minimally disturbs the response of the network to the previous training patterns [Widrow 87].

The initial belief that larger networks based upon the perceptron model could be trained to learn ever larger and more complex problems was shown to be unfounded with the proof [Minsky 69] that perceptrons were unable to represent, and hence learn, certain simple functions. One of the many examples used to demonstrate the limitations of the perceptron was the exclusive OR (XOR) function. Since it is not linearly separable, the perceptron could not learn this function, and correctly classify all four possible inputs. Minsky and Pappert's proofs of the limitations of single layer perceptrons, and their assertion that there

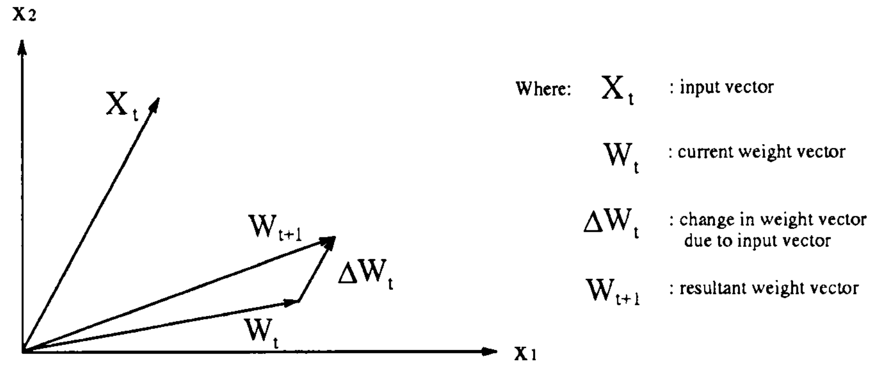

x2
X_t
W_{t+1}
ΔW_t
W_t
x1
Where: X_t : input vector
W_t : current weight vector
ΔW_t : change in weight vector due to input vector
W_{t+1} : resultant weight vector

Figure 1.5: Geometric Interpretation of Widrow-Hoff Weight Update Rule.

would be "no reason to suppose the existence of a learning theorem for a multi-layered perceptron" [Minsky 69] reduced the interest in neural network research for a number of years.

It was not until the development of the Back-Propagation (BP) algorithm, introduced by Rumelhart and McClelland in [Rumelhart 86a] (and later shown to have been previously developed in similar forms by a number of researchers, including Parker and Werbos [Hecht-Nielsen 90]) that a method for training multi-layer networks became available. On single-layer networks, the error associated with each pattern could be used directly to modify the weights contributing to that error. With multi-layer networks, this 'credit assignment problem' on hidden layer weights had not been possible. The BP algorithm relies on the use of a differentiable activation function (often a sigmoid function) which allows error terms to be propagated back through the network in order to be distributed proportionately among the weights contributing to that error. With each update, the weights are modified such that the overall system error is reduced. In essence, then, BP is a gradient descent algorithm. With the ability to train hidden layers, some of the limitations inherent in single layer networks were eliminated. Assuming a sufficient number of neurons exist, multi-layer networks can form arbitrarily complex decision regions as opposed to the linear decision boundaries which are formed by single-layer networks [Lippmann 87a]. The generalized delta rule for back-propagation of error and weight update is presented in Chapter 2.

1.4 Neural Network Models

In addition to the feed-forward, multi-layer networks trained with the back propagation algorithm, a number of other neural network models have been developed. They differ in many aspects (Section 1.2), including organization and function of processing nodes, as well as in their training algorithms. The Hopfield network is a single-layer fully connected recurrent (or feedback) network that can function as a content addressable memory. The Hopfield net can also be viewed as a non-linear dynamical system. Associated with this system is an energy function which can be minimized as the system evolves with time. This concept has led to a number of approaches for global optimization using neural networks, including the Boltzman machine [Ackley 85],[Hinton 86], and the Cauchy machine [Szu 87]. Networks with unsupervised training algorithms include the Kohonen Self-Organizing Feature Map and Carpenter/Grossberg's Adaptive Resonance Theory (ART) networks. These are described in more detail below.

The Hopfield network [Hopfield 82] uses binary inputs and outputs. All processing nodes receive input from all other nodes in the network. An initial input is applied to the system and each node produces an output which is the thresholded result of the sum of the product of the input signals and the input line's weight values. The outputs (high or low) of the nodes are then fed back into all other nodes (each node ignores its own output) as inputs again, and the process is repeated until the system stabilizes. The network always adjusts its weights in order to reduce the error, which is calculated as the difference between the actual output and the desired output. It thus descends the error surface until a local minimum is reached, and is guaranteed to do this in a finite number of steps, provided that the inputs are symmetrically connected, and the units are updated sequentially [Hopfield 82]. A Hopfield net can function as an autoassociative memory. A number of different patterns may be stored in its memory. Whenever one of the stored patterns is presented to the network, the memory will remain stable in that state. This is also known as a content addressable memory. If a distorted or noisy pattern is presented, the network will evolve from a representation of the input pattern to the stored pattern most closely matching it.

The Boltzman machine may be seen as a stochastic version of a Hopfield network [Ackley 85] in that a simulated annealing algorithm is incorporated into the processing at each node. Simulated annealing [Kirkpatrick 83] takes its name from the similarity to metallurgical annealing in which a metal is heated and allowed to slowly cool to a lower temperature. At relatively high temperatures, dislocations in the crystal lattice tend to reform into the normal crystal structure (with a resultant lowering of the crystal's energy level). These dislocations do not regenerate with suitably slow cooling. With no internal dislocations, the metal will have reached a global minimum with respect to its energy function.

The mechanism of 'simulated' annealing is to change a randomly selected point (atom) in the system at each iteration. If the change results in a lower overall error, the change is accepted. If a higher error level occurs, the change is accepted with a probability which is determined by the change in error divided by the 'computational' temperature, with a suitably chosen coefficient k, as follows:

$$p = \exp \frac{-\Delta Error}{kT}$$

Thus an increase in error is accepted with higher probability at high temperatures, but with much lower probability at lower temperature ranges. This allows escape from local minima to occur.

A neural network implementation of the simulated annealing algorithm uses the misclassification error as its energy level. The weights correspond to the atoms in the metallurgical model. T is typically set inversely proportional to the iteration count. Changes in weights which produce a reduction in system error are accepted, while those that produce an increase in error are accepted with a probability dependent upon the magnitude of the error increase, and the value T. Weight changes which cause increases in the system error similarly allow escape from local minima. In the Boltzman machine, the temperature can be lowered when either the energy in the system has decreased, or after a specified number of updates have been made. The effect on rates of temperature reduction and subsequent convergence of the simulated annealing algorithm is found in [Geman 84].

The so-called 'Cauchy training' method is another stochastic optimization technique. It is described by Szu and Hartley in [Szu 87], and uses the Cauchy distribution for calculating the step size for the weight adjustments. The broader distribution increases the probability of larger step sizes. This, in turn can lead to a reduction in training times versus the Boltzman machine. However, Wasserman notes in [Wasserman 89] that this feature may also allow excessive weight changes to occur which force a neuron into the state of producing a very high or very low activation level, which can cause subsequent weight changes to be near-zero. If a large number of nodes are affected, training may come to a virtual standstill, which he terms network paralysis.

The Kohonen self-organizing feature maps [Kohonen 84] used for pattern recognition are based on an unsupervised learning technique. The feature maps classify an input pattern (vector) based upon its similarity to an existing weight vector. The application of an input vector produces a response from each neuron in the network. The neuron with the highest degree of similarity represents the resultant classification of the input. The neuron's weight vector is then updated to reflect the inclusion of the newly classified input vector. The resultant weight vector thus represents an approximate center of mass of all input vectors assigned to that class. The actual location of each resultant class on the feature map is not known before training. Kohonen uses a neighborhood concept in which all nodes within a certain distance of the neuron closest in similarity to the input are updated. As training progresses, the size of the neighborhood is reduced until only a single neuron is modified with the application of each input vector. Kohonen reports a high degree of accuracy in phoneme classification using this technique [Kohonen 89].

The Adaptive Resonance Theory (ART) described in [Grossberg 88] is another unsupervised learning technique developed by Grossberg and Carpenter. ART is a complex model which attempts to avoid several problems encountered with other network models. In most other networks, the process of learning a new pattern once a network is trained may result in such a change to existing weight values that previously learned patterns may be lost. This could require a complete retraining of the network before the new pattern is recognized along with all the others. This is potentially a costly process, particularly in a rapidly changing environment.

The ART network and learning algorithm retain patterns which have previously been learned, yet also allow for new learning to take place. In simple terms, ART classifies an input vector by calculating an inner product (a measure of similarity) between the input and all stored classes. The class exemplar which is most similar to the input will produce the largest response and inhibit all other neurons. However, before the assignment of the input vector takes place, it must match the stored pattern to within a tolerance criterion, called the vigilance parameter. If the input pattern fails the tolerance test, it is considered (and stored as) a separate class. In this way, an input pattern cannot modify a stored class exemplar unless it matches to within a specified tolerance. While protecting itself from 'unlearning', ART can still learn new input patterns. Those inputs which fail the vigilance criterion merely become new classes, which in turn may compete against the others for 'ownership' of subsequent input patterns. The setting of the vigilance parameter is thus of great importance. If it is set too restrictively, all input patterns become separate classes, and if set too loosely, all inputs may be clustered together in one class.

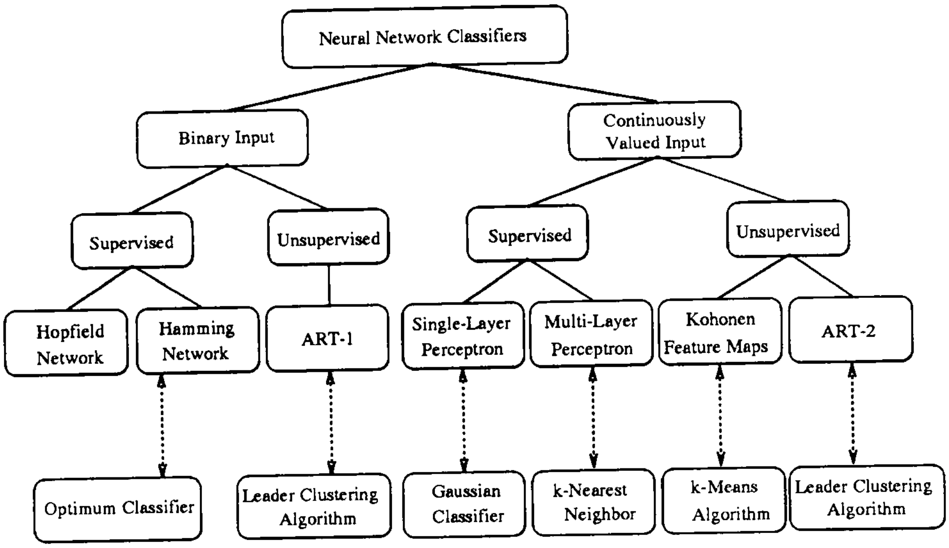


Figure 1.6: Taxonomy of Several Neural Network Models and Relation to Classification Algorithms

1.5 Neural Network Taxonomy

The different neural network models may themselves be classified based upon their classification techniques. A taxonomy of six important network models from [Lippmann 87a] is given in Figure 1.6, and described below.

The neural networks are organized into two categories: those which process binary input, and those which use continuously valued input. They are further separated into groups by the use of supervised or unsupervised training techniques in the implementation of their learning rule. The neural network models are then related to the classical pattern classification techniques to which they are most similar.

The Hopfield Net operates on binary inputs using supervised learning. It is an optimal minimum-error classifier [Lippmann 87a], if the class with the minimum Hamming distance to the input is selected. The Hamming distance is the number of bits which do not match between the input and class exemplar.

The Adaptive Resonance Theory networks, now known as ART-1 (using binary inputs) and ART-2 (using continuously valued inputs), are capable of learning without supervision. They are, however, sensitive to initialization of the critical vigilance parameter. While ART is a much more complex algorithm than the leader clustering algorithm, its basic operation is similar to it [Lippmann 87a]. The first input vector becomes the exemplar for the first class, and subsequent inputs are clustered together if they satisfy the tolerance criterion for closeness. The measure of closeness of an input vector to each stored exemplar can be produced by a dot product of the two vectors.

The Kohonen Self-Organizing Feature Map is another unsupervised learning algorithm which operates on continuously valued inputs. This model is similar in methodology to the k-means clustering algorithm, in which input vectors are assigned to a class based upon similarity of the input to each class vector.

The Perceptron model, operating on continuously valued inputs under supervised learning is capable of forming decision regions similar to the traditional maximum likelihood Gaussian classifiers. Whereas single-layer perceptrons can classify only linearly separable data, multi-layer perceptrons can generate more general decision regions. In particular, a two layer perceptron can form convex decision regions, but cannot classify correctly when the input classes are meshed, or surround one another. Three-layer perceptrons with sufficient nodes are able to form arbitrarily complex decision regions.

1.6 Goals for this Thesis

With the widespread use of the back-propagation algorithm in training multilayer, feed-forward neural networks, a number of heuristics have emerged. These have been designed to alleviate the major problems encountered with the algorithm, which are the long training times and convergence to local minima. In this thesis, the 'standard' BP algorithm from Rumelhart and McClelland [Rumelhart 86a] is compared against several of these heuristics, including the Gradient Reuse Algorithm [Hush 87], the Delta-Bar-Delta Algorithm [Jacobs 88], the Extended Delta-Bar-Delta Algorithm [Minai 90], modification of the sigmoid slope coefficient proposed in [Izui 90], and a replacement of random initial weights with values which pre-partition inputs into separate decision regions [Oblow 90].

The goals for this thesis are to test the different heuristics with three well known benchmark problems: the XOR (parity), the multiplexer, and the encoder/decoder problems, which are described in detail in Chapter 3. These three problem types are different in character and present different levels of difficulty to the network. The XOR (parity) predicate requires a network to make sharp distinctions between similarly appearing input, the multiplexer represents a problem in which a network must recognize a relationship among the inputs, and the encoder/decoder problem measures a network's ability to generalize.

Initially, a baseline map of convergence rates for the standard back-propagation algorithm was generated to which the results of the heuristics were compared. The heuristics were run on small problems in order to determine the effect that parameter changes have on convergence rates. The three problem types were then increased in size, in order to better understand the scalability of the heuristics. The XOR problem was run with n=2, 3, and 4 inputs using a n-n-1 (fully connected) network architecture. The multiplexer problem, also using a n-n-1 architecture was run with n=3, 6, and 11 inputs. The encoder/decoder problem used an n-m-n architecture where n is typically a power of 2 and m=log(n). In these trials, networks of size 4-2-4, 8-3-8, and 10-5-10 were tested. As the size of the problem was increased, the corresponding change in convergence times was monitored to determine the effect on the performance of the algorithms.

Chapter 2

Back-Propagation Algorithm

2.1 Description of Algorithm

The back-propagation algorithm was developed for training weights on multilayer, feed-forward neural networks with a non-decreasing and differentiable activation function. This development allowed the hidden layers of neural networks to be trained directly. The algorithm uses information local to each neuron to iteratively modify weights in order to minimize error at the output units. As a gradient descent procedure, BP is not guaranteed to find the global error minimum.

The BP algorithm is a supervised learning rule which involves the presentation of a set of input and output patterns. For each applied pattern, two phases of processing occur. The first phase is the forward pass, in which the network response to the input is determined, and an error term reflecting the difference from the desired output is calculated. Phase two consists of the weight adjustment pass, in which the error is fed back through the network layers in order to modify the weights such that the overall error is reduced. It is essential that the non-linear activation (usually a sigmoid) function be differentiable. Its derivative is necessary for the weight adjustment calculation. In the following description of BP, i,j,k are indices for nodes (and their associated weights) in the input, hidden, and output layers, respectively (Figure 1.1). The index, i, may range in value from 1 to n, where n is the number of input nodes, j ranges from 1 to m, where m is the number of hidden layer nodes, and k ranges from 1 to K, where K is the number of output layer nodes. The index, p, is the input pattern index and ranges in value from 1 to P, the total number of input patterns contained within the training set. The process by which a single hidden layer is trained with BP is described here, however, the procedure is applicable to multiple hidden layers.

The forward pass consists of the following steps: the input pattern is applied to the input layer, which fans the inputs out to each of the nodes in the hidden layer. For each node j in the hidden layer, the activation level net_{pj} is calculated as the weighted sum of the inputs resulting from the application of the p^{th} pattern to the input layer nodes:

$$net_{pj} = \sum_i w_{ij} x_{pi} + \theta_j$$

The term θ_j serves as a threshold value. The output, o_{pj}, of each node in the hidden layer is calculated by applying a sigmoid activation function f to the activation level, net_{pj}. For example:

$$o_{pj} = f(net_{pj}) = \frac{1}{1 + \exp^{-net_{pj}}}$$

The hidden layer outputs are, in turn, fanned out and used as inputs to nodes in the output layer. The activation level, net_{pk}, and output, o_{pk}, for each node in the output layer are calculated in a similar fashion:

$$net_{pk} = \sum_j w_{jk} o_{pj}$$

$$o_{pk} = f(net_{pk})$$

The error at each node k in the output layer which is produced by pattern p is calculated by comparing the output at each node with the corresponding element of the desired output vector.

$$E_{pk} = t_{pk} - o_{pk}$$

The second phase consists of the back-propagation of this error term and the subsequent weight adjustment. The derivative of the sigmoid function is used in this pass. As shown in Appendix A the partial derivative of the sigmoid function with respect to its input is:

$$\frac{\partial o_{pk}}{\partial net_{pk}} = f'(net_{pk}) = o_{pk}(1 - o_{pk})$$

For each neuron k in the output layer, an error, δ_{pk}, is calculated for each pattern in the training set that is applied to the network:

$$\delta_{pk} = f'(net_{pk})\ E_{pk} = o_{pk}(1 - o_{pk})(t_{pk} - o_{pk})$$

The error signal, δ_{pk}, from nodes in a layer is fed back through the weights on the links connecting the preceding (hidden) layer to this one, modified by $\frac{\partial o_{pj}}{\partial net_{pj}}$, to produce an error term, δ_{pj}, associated with each node j in the preceding hidden layer:

$$\delta_{pj} = o_{pj}(1 - o_{pj}) \sum_k \delta_{pk} w_{jk}$$

Using these error terms, the change in weights caused by the difference in the desired response of the network to its actual output as a result of the application of the p^{th} training pattern may be calculated for both output and hidden layer nodes:

$$\Delta_p w_{jk} = \eta \delta_{pk} o_{pj}$$

$$\Delta_p w_{ij} = \eta \delta_{pj} x_{pi}$$

This process is repeated for each pattern in the training set. After the application of all patterns in the training set, known as an epoch, the square of the pattern level error is summed:

$$E_p = \frac{1}{2} \sum_k (t_{pk} - o_{pk})^2$$

This is a measure of squared error from all output nodes resulting from the application of one pattern. The total system error, which is an overall measure of network performance, is the sum of all pattern error terms:

$$E_{TOT} = \sum_p E_p$$

The per pattern weight changes are summed over all input patterns in the training set, producing, at the end of an epoch, a single weight change for each weight and threshold term:

$$\Delta w_{jk} = \sum_p \Delta_p w_{jk}$$

$$\Delta w_{ij} = \sum_p \Delta_p w_{ij}$$

The hidden layer weights and the output layer weights are updated with these Δw terms after each complete pass through the training set, shown as follows for the w_{jk} weights, but calculated similarly for the w_{ij} weights, as well:

$$w_{jk}(t+1) = w_{jk}(t) + \Delta w_{jk}(t)$$

Combining terms, the weight update equation becomes:

$$w_{jk}(t+1) = w_{jk}(t) + \eta \sum_{p} \delta_{pk} o_{pj}$$

These weights are then used to generate classifications for patterns in the subsequent epoch.

The initial weight values are usually randomly assigned. Rumelhart, Hinton and Williams, in [Rumelhart 86a], note that if all weight values are assigned the same value initially, no learning can occur because the error is propagated back through the weights in proportion to their values. If they are all set identically, they will always be modified by the same amount, and hence, will never change with respect to each other. Thus, no distinction between inputs can take place. It is also mentioned that the weight update expression can be modified to include a momentum term, α, such that:

$$\Delta w_{jk}(t) = \eta \sum_{p} (\delta_{pk} o_{pj}) + \alpha \Delta w_{jk}(t-1)$$

The momentum coefficient thus determines the effect that past weight changes will have on the current direction of change in weight space. The momentum term helps filter out the high frequency variations of the weight update term which can cause oscillatory behavior in the network response. If only small variations exist, the momentum term tends to increase the Δw adjustment, thus moving the system more quickly towards a minimum. Unfortunately, overstepping the minimum is also more likely to occur, as well. The momentum term, α, is normally set to values within the range $0 \leq \alpha < 1.0$. The use a value of zero disables momentum, and results in the original weight update rule.

2.2 The Gradient Reuse Algorithm

The Gradient Reuse Algorithm (GRA), as defined by Hush and Salas [Hush 88], is a simple extension to the standard back-propagation algorithm which may lead to improvements in convergence rates. The GRA attempts to speed up BP in two ways. First, the error gradients used to determine weight changes are reused until either the weight updates no longer result in a reduction of error, or a limit is reached on the number of reuses of the gradient. Reusing gradients to update the weights, on average, reduces the number of gradient calculations which must be made, and in effect, increases the step size coefficient. Second, the gradient reuse rate is employed to modify a convergence parameter, μ. This parameter is an adaptable version of η, the learning rate parameter from the standard BP algorithm.

The gradient term, $\Delta_p g_{ij}$, associated with the weight, w_{ij}, is calculated for each pattern, and summed over all patterns to obtain the gradient term, g_{ij} (g_{jk} terms are calculated in similar fashion):

$$\Delta_p g_{ij} = \delta_{pj} x_{pi} \quad \Delta_p g_{jk} = \delta_{pk} o_{pj}$$

$$g_{ij} = \sum_p \Delta_p g_{ij} \quad g_{jk} = \sum_p \Delta_p g_{jk}$$

These gradients, formed at epoch t are used to calculate the weight change ($\Delta w_{jk}(t+1)$ is similarly calculated):

$$\Delta w_{ij}(t+1) = w_{ij}(t) + \mu g_{ij}(t)$$

As indicated, the weights are updated after each pass through the training set. The convergence parameter μ is also updated each pass. μ is increased (resulting in larger subsequent weight changes) whenever the reuse rate is high and it is decreased whenever the reuse rate is low. These lower and upper limits on reuse rates were set to 5 and 10 respectively by Hush and Salas [Hush 88]. The advantage to this adaptive technique is that the learning rate parameter is adapted differently for different areas of the error surface. Whenever the error curve is relatively flat (error gradients near zero), many gradient reuses would result in increases to the learning rate parameter, thus increasing the step size and reducing the number of subsequent steps required. Correspondingly, whenever the error surface is relatively steep, fewer reuses would result in a decrease to the learning rate parameter, thus producing a smaller step size. In this way, the dynamic convergence parameter helps prevent the search from stagnating on flat portions of the error surface, while maintaining the accuracy of the search on the steeper portions of the error surface [Hush 88].

The gradient is reapplied as long as the overall error is reduced. Thus, the GRA may be viewed as a line search technique, since a search is performed for a minimum in the direction of the gradient [Hush 88]. The search is stopped at a point where an overall increase in the error in the direction of the gradient occurs.

2.3 The Delta Bar Delta Algorithm

The Delta Bar Delta Algorithm (DBD) defined in [Jacobs 88] is a method based upon four heuristics for achieving faster rates of convergence compared to the standard BP algorithm. The slow convergence of the original algorithm was thought to be due to a number of reasons. These include:

- On a flat error surface, the magnitude of the partial derivative of the error with respect to each weight may be so small that repeated weight adjustments must be made before any significant reduction in error can occur.
- On a highly curved error surface, a large weight adjustment may be made, allowing a minimum in that weight dimension to be passed over.
- The direction of the negative gradient may not point directly towards the minimum of the error surface.
- A constant learning rate may not be suitable over all areas of the error surface. Learning rates which produce reasonable steps in one weight dimension may not be suitable in another.

These problems led to the development of the following four heuristics in [Jacobs 88]:

- Every parameter to be minimized has its own individual learning rate.
- Every learning rate is allowed to vary.
- When subsequent derivatives of the error surface with respect to a weight possess the same sign, the learning rate increases.
- When subsequent derivatives possess different signs, the learning rate should be decreased, as a minimum is being stepped over.

Jacobs further states that by providing different modifiable learning rates for each weight dimension, the current point in weight space is not modified in the direction of the negative gradient. Thus, the system is not performing true gradient descent [Jacobs 88].

The Delta Bar Delta algorithm implements these four heuristics. It consists of both a weight update rule and a learning rate update rule. The weight update rule is similar to BP, except that each weight possesses its own modifiable learning rate parameter:

$$w(t+1) = w(t) - \epsilon(t+1)\frac{\partial J(t)}{\partial w(t)}$$

where $\epsilon(t)$ is the learning rate associated with weight $w(t)$, and $\frac{\partial J(t)}{\partial w(t)}$, which is the partial derivative of the error with respect to $w(t)$, is written as $\frac{-\partial E(t)}{\partial w(t)}$ in the standard back-propagation algorithm. From Rumelhart, et al [Rumelhart 86a],

$$\frac{-\partial E(t)}{\partial w(t)} = \delta_{pj} o_{pi}$$

Thus, the DBD weight update rule is the same as the back-propagation weight update rule except for the weight specific adaptable learning rate coefficient, ϵ replacing the constant learning rate η (shown for w_{jk}):

$$w_{jk}(t+1) = w_{jk}(t) + \epsilon_{jk}(t+1)\delta_{pk} o_{pj}$$

The mechanism for adaptively modifying the step size parameter is contained within the learning rate update rule from [Jacobs 88], defined as:

$$\Delta\epsilon(t) = \begin{cases} \kappa & if\ \ \bar{\delta}(t-1)\delta(t) > 0 \\ -\phi\epsilon(t) & if\ \ \bar{\delta}(t-1)\delta(t) < 0 \\ 0 & otherwise \end{cases}$$

where $\delta(t)$ is the partial derivative of the error with respect to $w(t)$:

$$\delta(t) = \frac{\partial J(t)}{\partial w(t)}$$

and $\bar{\delta}(t)$ is the exponentially decaying trail of the current and past derivatives with θ used to determine the impact of the past derivatives:

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

There are three additional parameters which must be used in the algorithm: κ, ϕ, and θ. κ is used to increase the learning rate for a weight if the current derivative of the error for that weight, $\delta(t)$, and the exponential average of the weight's previous derivatives, $\bar{\delta}(t)$ possess the same sign. If $\delta(t)$ and $\bar{\delta}(t-1)$ possess oppposite signs, the learning rate for that weight is reduced by a proportion, ϕ, of its value. $\bar{\delta}(t)$ is then updated using the current $\delta(t)$ and its past value, $\bar{\delta}(t-1)$, as shown in the above equation.

The DBD rule thus allows learning rates to be modified with linear increases, which tends to reduce the number of steps taken, but does not allow them to become too large too quickly. It further allows exponential decreases, which allow for quick reductions in the learning rate when a minimum is being passed over.

2.4 The Extended Delta Bar Delta Algorithm

The Extended Delta Bar Delta (EDBD) algorithm from Minai and Williams [Minai 90], is a modification of the DBD algorithm just presented. Minai notes the following limitations with the DBD algorithm as it is defined:

- The DBD did not use momentum as in BP. Although Jacobs acknowledged the use of momentum as beneficial, it was not included in the definition.

- The learning rate update constant, κ, sometimes caused such large increases over time that even small exponential decreases were not enough to prevent excessively large weight updates.

The following heuristics were developed to avoid these problems with the DBD algorithm. They form an extension to the DBD algorithm in that they use as their foundation the DBD definition. These heuristics are:

- The learning rate increase is changed from being linear to an exponentially decreasing function of $|\bar{\delta}(t)|$. This produces faster increases on flat areas, but slower increases on steeper error slopes.

- Momentum is used, and varied similarly to the learning rate.

- An upper limit is imposed for both learning rate and momentum.

- Memory and recovery are implemented. If the overall error becomes greater than λ times the lowest error achieved, the search is restarted at that point (weights reset to the values which produced the lowest error) with a reduced learning rate and momentum term. The weights are reset with some probability of change in order to prevent the algorithm from looping continuously back to the same minimum error weight terms each time it failed to improve upon them.

The EDBD algorithm implementing these heuristics consists of a weight update rule, a learning rate update rule, and a momentum update rule. The specific algorithms used to implement the last of the heuristics above regarding memory and recovery were not defined by Minai, and were not implemented in his tests. They were similarly not implemented for these tests. The weight update equation for the EDBD algorithm is:

$$\Delta w_{ij}(t) = -\eta_{ij}(t)\frac{\partial E(t)}{\partial w_{ij}(t)} + \alpha_{ij}(t)\Delta w_{ij}(t-1)$$

which is the same as the standard BP algorithm, except that it uses an adaptive stepsize and momentum coefficient for each weight. The learning rate and momentum update rules are:

$$\eta_{ij}(t+1) = min[\eta_{max}, \eta_{ij}(t) + \Delta\eta_{ij}(t)]$$

$$\alpha_{ij}(t+1) = min[\alpha_{max}, \alpha_{ij}(t) + \Delta\alpha_{ij}(t)]$$

where min is the function returning the minimum of its arguments, and α_{max} and η_{max} are the maximum allowable level for learning rate and momentum. $\Delta\eta_{ij}(t)$ is defined as:

$$\Delta\eta_{ij}(t) = \begin{cases} \kappa_l \exp^{(-\gamma_l|\bar{\delta}(t)|)} & if \;\; \bar{\delta}(t-1)\delta(t) > 0 \\ -\phi_l \eta(t) & if \;\; \bar{\delta}(t-1)\delta(t) < 0 \\ 0 & otherwise \end{cases}$$

and $\Delta\alpha_{ij}(t)$ is defined to be:

$$\Delta\alpha_{ij}(t) = \begin{cases} \kappa_m \exp^{(-\gamma_m|\bar{\delta}(t)|)} & if \;\; \bar{\delta}(t-1)\delta(t) > 0 \\ -\phi_m \alpha(t) & if \;\; \bar{\delta}(t-1)\delta(t) < 0 \\ 0 & otherwise \end{cases}$$

where $\delta(t)$ is the partial derivative of the error with respect to $w(t)$:

$$\delta(t) = \frac{\partial J(t)}{\partial w(t)}$$

and $\bar{\delta}(t)$ is the exponentially decaying trace of the current and past derivatives with θ as the base and the iteration t as the exponent:

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

$\delta(t)$ and $\bar{\delta}(t)$ are calculated exactly as specified in the DBD algorithm. The parameters thus required for the implementation of the EDBD algorithm are: $\kappa_l, \gamma_l, \phi_l, \kappa_m, \gamma_m, \phi_m, \theta, \eta_{max}, \alpha_{max}$, and λ (see Symbol Table).

2.5 Modification of Sigmoid Shape

The non-linear activation function used in BP is normally defined as the sigmoid function, f, for example:

$$out = f(net) = \frac{1}{1 + \exp^{-Dnet}}$$

where *out* is the output of the sigmoid function resulting from an activation level of *net*, and D determines the slope (or 'sharpness') of the sigmoid function. In the BP algorithm, D is assumed to be 1. Izui and Pentland [Izui 90] report an improvement in convergence times using a slope coefficient greater than 1.0.

The derivative of the sigmoid function is [Appendix A]:

$$f'(net) = out(1 - out)$$

The derivative reaches a maximum of 0.25 at an *out* level of 0.5, and approaches its minimum of 0.0 as *out* approaches 0.0 or 1.0. The amount a weight is changed is affected by the value of the derivative. Weights for neurons that are near their midpoint will be changed most. Thus, neurons which are 'uncommitted' are modified the most, while neurons which are generating an output near the limit are modified least. Rumelhart Hinton, and Williams in [Rumelhart 86a] suggest that this feature contributes to the stability of the learning of the system by limiting subsequent weight change to neurons which have settled to a definite output state.

For $D \neq 1$, we obtain:

$$f'(net) = D \cdot out(1 - out)$$

This modifies the performance of BP in two ways. In the forward pass, the output generated by each node is altered by the change to the sigmoid activation

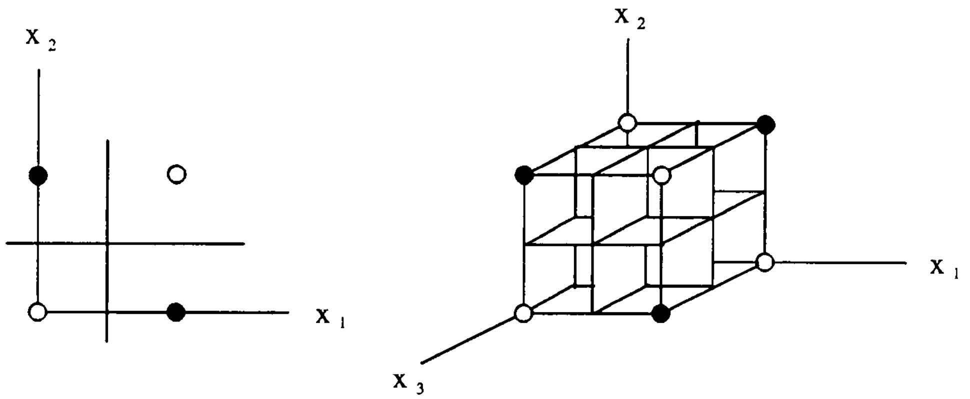


Figure 2.1: Decision Boundaries in 2 and 3 Dimensions Produced Using Preset Weights.

function, resulting in output activation levels more extreme than before. In the reverse pass, the weight adjustment is magnified by the coefficient D. Both measures tend to increase the size of weight updates.

2.6 Initial Weight Modification

The final variation of BP considered in this work is an adjustment to the random assignment of initial values given to all weights. Instead of forcing a neural network to begin operating from a randomly assigned starting point, it was conjectured [Oblow 90] that the performance of the network could be improved by setting the initial weights to values which partitioned the input parameter space such that each input pattern would lie in a separate decision region (Figure 2.1). The weights between the hidden and output layers would then be required to learn to develop a representation which would associate these decision regions in order to output the correct classification. The reduction in the complexity of the problem suggested that faster convergence rates might result.

In general, this concept would require pre-processing all input data to determine the exact locations of each of these 'grid lines', in order to calculate the appropriate weight values. With binary inputs, this was not required. For the XOR and multiplexer problems (described in Chapter 3), with n binary inputs, the n-dimensional hypercube was divided into 2^n regions using n mutually orthogonal hyperplanes, such that each vertex would be found in a distinct decision region. Since each of the 2^n inputs can be viewed as being situated on a specific vertex of the hypercube, each input would then be in its own individual region.

For single-layer networks the application of a pattern with n inputs through the weights to an input node had the effect (if interpreted geometrically) of generating a linear decision boundary (see Figure 1.4): a line in two dimensions (n=2), a plane in three dimensions (n=3), and a hyperplane in the n-dimensional case. As additional inputs were applied to the network the weights were adjusted such that inputs from different classes were separated from each other by this hyperplane. Each hyperplane could divide a single parameter dimension into two regions. If the hidden layer weights could be pre-set to divide input parameter space into multiple cells, each cell containing at most one input, then the training time required to generate these separating hyperplanes could be eliminated. The network would be required to associate the decision regions in order to reduce classification error.

With binary data, a boundary midway between 0 and 1 was chosen in each of the n dimensions. In two dimensions this corresponds to specifying weights which form linear boundaries at $x_1 = 0.5$, and $x_2 = 0.5$. In three dimensions, single plane boundaries are formed at $x_1 = 0.5$, $x_2 = 0.5$, and $x_3 = 0.5$. These decision boundaries for the two and three dimensional cases are shown in Figure 2.1. The initial weights for the input nodes are defined such that one of these boundaries is specified by the weights of one of the hidden layer nodes. This implies the need for an n-n-1 network architecture in order to produce the required number of hyperplane boundaries with n input lines. From the 2-dimensional case, we recall the input summation at a node, and the corresponding equation of the separating line thus formed:

$$x_1w_1 + x_2w_2 + \theta = 0$$

or in slope intercept form:

$$x_2 = -x_1\frac{w_1}{w_2} - \frac{\theta}{w_2}$$

To create a separating plane at $x_1 = 0.5$, the x_2 input is ignored by setting its corresponding weight, w_2, to 0.0. The relationship between the threshold, θ, and w_1 is defined by:

$$0.5 = -\frac{\theta}{w_1}$$

A selection of w_1 to 1.0, sets θ to -0.5. Thus, the hidden layer nodes will have their bias (threshold) terms set to -0.5, and one weight set to 1.0. This weight must correspond to a different input line for each of the nodes in order to provide a decision boundary in each parameter dimension. The i^{th} weight to the i^{th} node was chosen as 1.0. All other weights in the input layer were initialized to 0.0. This separated the n-dimensional hypercube into 2^n regions, each containing one of the 2^n inputs. The modification of the input weight values was made for both the XOR and the multiplexer problems.

Chapter 3

Description of Benchmarks

The problem types chosen as benchmarks for this study are the XOR/Parity problem, the multiplexer problem, and the encoder/decoder problem. The XOR problem was chosen because numerous other studies have used it as a benchmark. As Fahlman noted [Fahlman 88], however, the XOR/parity problem may be overemphasized in terms of measuring the capability of a network's ability to classify inputs. Classification utilizes a network's ability to form generalizations from the patterns it is trained with. Occasionally, sharp distinctions must be made between similar inputs, but more often, inputs which are close in pattern space are classified together. The XOR problem, however, penalizes generalization. Inputs which differ by a single bit must generate opposite outputs. In order to better test the ability of a network to generalize, the two other problem types have been included. The multiplexer problem appears to be a less difficult problem than XOR. For the multiplexer problem, some generalization of the input patterns is useful, yet it still requires that occasional sharp distinctions be made. The encoder/decoder problem is probably the least difficult of the three

3 Bit XOR			
x_3	x_2	x_1	Out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

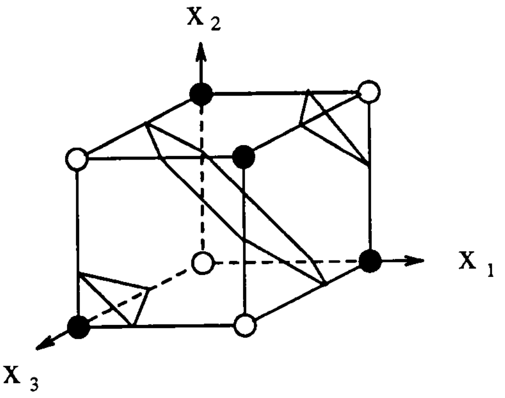


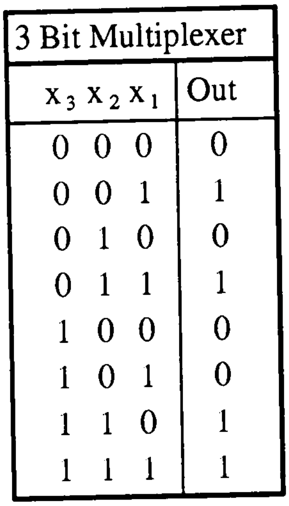
Figure 3.1: 3-Bit XOR/Parity Function Table and Graph with Output Classes Distuinguished with Separating Planes

problems, and presents an altogether different set of requirements to a neural network. These three problems represent three different types of classification problems, and represent three different levels of difficulty with which to test the algorithms used.

3.1 The XOR/Parity Problem

The XOR problem is one of the most common tests applied to neural networks. It was shown by Minsky and Pappert in [Minsky 69] to be unsolvable by a single layer perceptron. Multi-layer networks are not subject to the limitations of representations found in the single layer networks. XOR is one of two of the sixteen binary logic functions which do not lead to linearly separable classes [Wasserman 89]. The lack of linear separability explains the perceptron's inability to learn this function.

The XOR function may also be viewed as the two input case of the



3 Bit Multiplexer	
x_3 x_2 x_1	Out
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	1

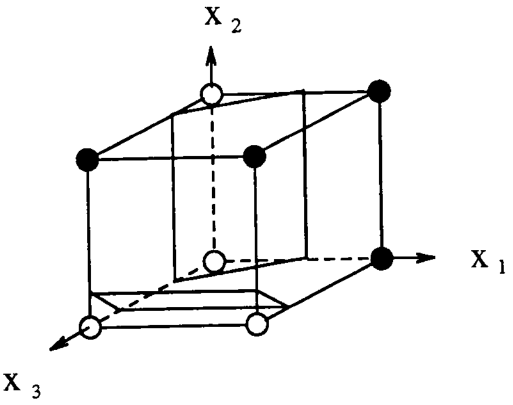


Figure 3.2: 3-Bit Multiplexer Function Table and Graph with Output Classes Distuinguished with Separating Planes

more general n-bit parity problem, in which the output is on ($= 1$) if an odd number of inputs are on, and off ($= 0$) otherwise. Inputs which differ by a single bit in their pattern must belong to different classes. The parity problem will be scaled up to 3 and 4 inputs. An n-n-1 network (for n input lines) will be used for the XOR/parity problem.

3.2 The Multiplexer Problem

The multiplexer problem consists of logically dividing a series of input lines into two types: address lines and data lines. Typically, the number of lines is chosen so that there can be n address lines and 2^n data lines. The binary address contained on the address lines is used to specify the data line of interest. The desired output of the system is the input to the data line addressed by the address lines. The 3-bit multiplexer problem (Figure 3.2), with 1 address line and 2 data lines, will initially be implemented, and compared to the 3-bit parity problem.

Input Lines											Desired Output
Address			Data								
a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	
0	0	0	0	0	0	0	0	0	0	1	1
0	1	1	0	0	0	0	1	0	0	0	1
1	0	1	1	1	0	1	1	0	1	0	0

Table 3.1: Examples of the 11-Bit Multiplexer Problem.

The 3-bit parity problem requires the network to form the equivalent of 3 hyperplanes in order to correctly classify all inputs, whereas the 3-bit mulitplexer requires only two (Figure 3.2). The convergence times for the 3-bit multiplexer will be used for comparison to the 3-bit parity problem, however, it appears that the multiplexer problem is a less difficult problem than parity. The multiplexer problem is scaled up to include the 6-bit (2 address, and 4 data lines) and the 11-bit (3 address, and 8 data lines), shown in Table 3.1. The network structure used on the multiplexer problems consists of an n-n-1 organization, where n is the number of input lines.

3.3 The Encoder/Decoder Problem

The encoder/decoder problem is generally implemented as a fully connected multi-layer network (Figure 3.4) with an n-m-n structure (n input and output units, and m hidden layer units), with 3 layers of nodes, and 2 layers of trainable

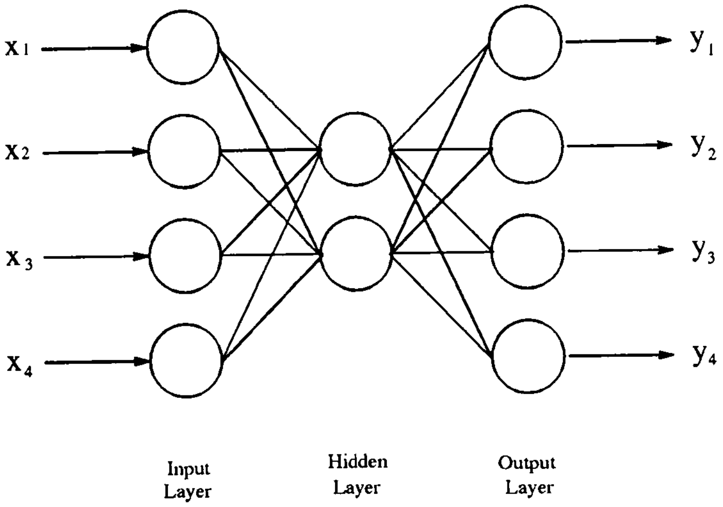


Figure 3.3: A 4-2-4 Encoder/Decoder Network.

weights. n is generally set as a power of 2 and m is usually defined as $m=log(n)$. [Ackley 85] A typical network structure might then be 4-2-4, or 8-3-8. The network is presented with n inputs, each of which has only one input line turned on (= 1). All other lines are turned off (= 0). The network is to be trained to produce the input pattern in the output nodes. With fewer than n nodes in the hidden layer, the network must create an internal encoding for all n inputs within the m hidden layer nodes that allows it to decode the correct output pattern given any of the n inputs. The encoder/decoder problem is run with sizes of 4-2-4, 8-3-8, and 10-5-10.

Chapter 4

Experimental Method

There does not appear to be a standard methodology for comparing the results of different learning algorithms. New learning rules have been tested with different benchmarks, different parameters, and different error criteria. All of these variables make it difficult to judge the relative performance of learning algorithms. The design of the tests used in this work was based in part upon the need for using a standardized means for comparison.

4.1 Experimental Design

A number of different methods for testing learning algorithms on fully-connected feed-forward neural networks have been used [Jacobs 88], [Fahlman 88], [Minai 90]. The need for standardized methods for comparison became apparent in three primary areas: benchmark selection, error criteria specification, and parameter specification.

The three benchmarks described in Chapter 2 were used to test the algorithms. Multiple problem types were selected in order to test the learning capabilities of the algorithms under different conditions. The selection of an error criterion is an important consideration. It is a specification that must be made prior to training the network, and is ultimately the criterion by which a network is judged to have successfully learned. It is also a somewhat difficult specification to make. In cases where imperfect information is used to train a network, or where noise cannot be removed from the data, zero error results may be impossible to attain. Different problems may demand different error tolerances, so an exact error specification cannot be made which would be suitable for all types of problems under all conditions. Fahlman, in [Fahlman 88] discusses several different criteria which have been used. These criteria are:

- a sharp threshold
- a small individual (per-pattern) error
- a small composite error
- winner-take-all
- threshold with margin

Fahlman notes the following problems with these error criteria: a sharp threshold, generally set such that any output greater than 0.5 is treated as a 1.0, and any output less than the threshold is treated as a 0.0, is natural in a binary system. Noise in the input values could affect the output state of a neuron, affecting the performance of the system. The small individual error criterion, where each output must be within a tolerance level of the desired output may be unnecessarily strict in a system with binary outputs because the outputs need only reflect a

0 or a 1. The small composite error criteria, such as the sum of squared errors requires that an error term be summed over all outputs over all patterns in a training set and still be no greater than the specified value. A problem with this method is that in a system with either a large number of outputs, or a large number of input patterns (or both), an incorrect output with relatively large error may be masked by a large number of outputs with low error. The winner-take-all criterion is useful whenever each of multiple outputs represents a distinct answer (such as with the encoder/decoder problem). In the case of near ties between two outputs, noise might cause an incorrect result. The threshold with margin criterion is similar to the sharp threshold method, except a margin is added in order to reduce the impact of noise on the correct firing of the output node.

The output of nodes with a sigmoid function cannot equal either 0.0 or 1.0, but may only approach these values asymptotically with either very low or very high activation levels. Since 0.0 and 1.0 are desired output values in these tests, and the output nodes cannot achieve these values, a desired error level of 0.0 would be impossible to achieve. An error level greater than 0.5 would indicate incorrect or incomplete learning. The selection, then, should be of an error level somewhere within this range. The lower the selected error level, the higher the level of assurance that the network has learned correctly. However if it is set too low, the network might require excessively long training times.

The methods used in this study included the sum of the squared error criterion, and the small individual error criterion, as defined above. The sum of the squared error, summed over all output lines, over the full training set is the method employed by Rumelhart & McClelland in both their definition of the BP algorithm [Rumelhart 86a], and in their simulator, as defined in [McClelland

86]. This method, therefore, was used in the initial series of tests. It was later modified to include a maximum individual error value because of problems caused by larger test cases. The problem was two-fold: as pattern sets became larger, the additional patterns caused a corresponding increase in the output error level, even though the per-pattern error level might be no different than for the small problem case. For example, in a test with 64 training patterns, the overall sum of the squared error would be roughly sixteen times the level seen on a test with 4 patterns, although the average error per pattern might be approximately the same. Merely allowing sixteen times the overall error level with sixteen times the number of input patterns may resolve this problem, but creates another problem which is even worse: it allows misclassification errors to be obscured. A single large error (due to incorrect classification) might be summed with many very small errors, resulting in an overall sum-of-squared-error less than the overall error level specified. This situation had to be avoided. Therefore, a maximum individual error criterion was implemented. The error value was specified on a per pattern basis. The sum of the squared error at each output node was not allowed to exceed this error level. Thus, the overall error level may be large (it will generally be larger for larger training set sizes and for networks with more output nodes), but each pattern will be correctly classified by all output nodes to within the specified tolerance level. This method can be implemented unchanged in networks with large numbers of training samples and/or with greater numbers of output nodes.

The error tolerance value used was 0.04. This level of error was chosen because it reflects a system state which has successfully trained its weights to correctly classify its inputs to within a small overall error level. Additional training could reduce the error further, but no change to the overall classification would be made. Of chief importance, is that with this level of error, no output nodes are in or near an 'undecided' state. All have been trained to produce correct responses with only a small degree of difference to the desired output for all training patterns. Any error value in the vicinity of 0.04 would also have been acceptable. There is not much difference between an error level of 0.04 and 0.05, for instance. This value was selected, because it was also used in some of the simulator runs in [McClelland 86]. Use of the same error criterion allowed results to be compared. Its use on all problem types permitted an overall evaluation of convergence times to be made.

The parameter settings which were used for making the runs have been recorded in order to relate performance to the initial conditions in effect. This is important not only in being able to interpret results from these tests, but also to be able to relate these results to others which are not so accurately catalogued.

Another important factor in measuring network success is that the initial weight files can cause large differences in convergence times [Kolen 90]. Random weights were created and stored in weight files for use by each algorithm. The same weight files were utilized by all methods on the same problem in order to be able to compare their results. As problem sizes were increased, network size increased, necessitating additional initial values for the additional weights. New files with random weight values were then created. Eleven weight files were established. This number of trials provided a reasonable set from which to calculate an average response for the network, yet was small enough so that a large number of parameter settings could be tested for each of the problem types. Each algorithm would use each of the eleven initial weight setups for a single set of runs. The results from the eleven runs would then be averaged. In the case of non-converging runs, the average of the runs which did converge is calculated, and the number of runs which failed is also reported. For larger problems, where simulation on the same large number of initial conditions on a sequential machine became impractical, five runs were used to determine an average overall network response.

It should also be noted that the weight update for all algorithms is based upon a complete pass through all patterns in the training set, which is called an epoch. This method of weight updating is termed 'batch' mode in [Rumelhart 86a], and is performed in the following fashion. Training patterns within the training set are presented to the network one at a time. For each pattern, the activations at each neuron are propagated forward to the subsequent layer, ultimately producing an output at each of the output units. This output is compared to the desired output, which is also specified as part of the training pattern. The difference is the error for that training pattern. This error is propagated back through the network and is divided up among the weights in proportion to their contribution to the error. The per pattern Δweight value is summed over all input patterns to arrive at a single Δweight value for each of the trainable weights in the network. At the end of each epoch, the weights are modified by their respective update value. In the algorithm with adaptive parameters, the update was also made at the end of each epoch.

After all algorithms were run on the 2-bit XOR problem, results could be compared. The algorithms were then run on the other two problem types, using as a starting point those parameters which performed well with the XOR case. The results of these tests provide a means for comparing the effectiveness of the different algorithms on problem types of varying complexity. The initialization of weight values was not implemented on the encoder/decoder problem. The idea behind this approach was to create n mutually orthogonal separating hyperplane specifications in the hidden-layer weights. This requires n input nodes and n hidden layer nodes. The encoder/decoder problem uses $m=log(n)$ hidden layer nodes, making this type of modification impractical to implement.

After baseline convergence times were determined for the algorithms on each of the benchmark problems, the problem sizes were increased. The XOR/Parity problem was run with the number of input lines, n, set to 3 and 4, also. The multiplexer problem was also run with n set to 6 and 11 in addition to the base case of 3 input lines. The encoder/decoder problem was then run on networks of size: 4-2-4, 8-3-8, and 10-5-10. The results of these runs provides information useful in determining the effects of scaling on convergence times.

Timing trials were also performed for each of the algorithms on each of the problems. These runs yielded an average per epoch computation time relative to the BP algorithm. This value, while not of great interest from the standpoint of parallel implementation, provides, when coupled with convergence times, another performance measure on sequential machines. It is a measure of the exponential increase in convergence times as measured in non-parallel implementations. This may be useful in determining approximate execution times as larger and more complex problems are tested on sequential machines.

4.2 Implementation

It was initially believed that the simulator, available with C source code in the McClelland and Rumelhart lab manual [McClelland 86] would be used to implement the algorithms and modifications. After becoming familiar with its operation, some modifications were made to the source code to begin the process of implementing the different algorithms. It quickly became apparent that due to the fairly complex changes that were required by the different algorithms, that these changes and additions would be more easily implemented if all routines were collectively designed and developed. The code was written in C and initially tested against the simulator for corresponding 2-bit XOR problems using the same weight files. Identical results (to whatever resolution used) were recorded. This includes per pattern error values, weight modification values and error gradient terms. Convergence times and error at convergence were identical for the several comparison tests which were run. The code was subsequently ported from an IBM-AT class machine to a Sun-3 workstation with no major code changes, with the same results noted on comparable runs, also. The porting of the C code to a more powerful machine was necessary in order to achieve results on the larger-sized problems (4-bit Parity, and 11-bit Multiplexer) in a reasonable period of time. Additions were then made to incorporate each of the different heuristics into a separate program. Each program accepted as command line arguments values for the parameters specified in the description of each algorithm (in Chapter 2). Each program then made sequential runs using these parameter settings on each of the weight files. For each initial setting, the patterns in the training set were sequentially applied to the inputs of the network, the resultant output of the network was compared to the desired output, and the difference was squared. This squared error was then back-propagated (as described in Chapter 2.1) to produce a per-pattern weight adjustment. These values were summed over all patterns and a single weight adjustment value was the result. This value was then multiplied by the learning rate coefficient (step size) and if momentum was employed, the momentum coefficient was multiplied by the previous Δweight value and added in. All weights and bias terms were similarly updated at the end of each epoch. At this time, the adaptive terms in the different algorithms were updated as well, as defined by their individual update rules as specified in their respective sections in Chapter 2. The network was allowed to iterate through epochs of training data until the error tolerance level was achieved, or until it had converged to a non-optimal solution at a local minimum. If successful, the number of epochs required for convergence was retained, and the cycle was repeated for the next set of input weights: parameters were reset to the original command line arguments, weights initialized to the values specified in the next weight file, and the training samples were reapplied sequentially to the input nodes. This was done until all 11 (or 5, for the larger problem sizes) weight files had been processed. After the initial settings had been tested, the results for the runs were averaged. At any time during processing, the execution of the program could be interrupted and the intermediate results of calculations examined. Per-pattern error gradients and error outputs could be monitored. This was done very often initially, in order to better understand the details of the weight adjustment phase, and also to better understand the behavior of the system as it became trapped in local minima.

If a run did not converge within a prescribed number of epochs to a solution within the error criteria, it was stopped and recorded as a failure. At the end of all runs for a particular set of parameter settings, two numbers could be used to describe the performance of the network: a ratio of converging runs to total runs made, and an average convergence time (measured in epochs) for the converging runs. This process was then repeated many times with a different set of parameter values, then repeated for each of the algorithms tested.

Setting a good a priori maximum limit on the number of epochs for a particular run was occasionally a difficult task, particularly for a new algorithm with untested parameters. Small parameter changes could cause large changes in convergence times, often of an order of magnitude or more (see results in Chapter 5). This presented a problem in that potentially convergent runs could be terminated prematurely. If the maximum limit was set too large, much time would be spent on convergence to non-optimal solutions. The determination of the maximum limit to use depends upon the type of problem being processed, the size of the problem, and the initial conditions being used (including both weight and parameter settings).

For the BP algorithm, learning rate was varied over the range $0.0 < \eta \leq 1.0$. The momentum was varied over the range $0.0 < \alpha \leq 0.95$. After testing these combinations in 0.1 increments, larger learning rates were examined. The initial learning rate parameter and the learning rate update increment are varied in the GRA in order to determine the effect of initial parameter settings to convergence rates. The sampling of large numbers of parameter combinations became impractical with the increase in the number of modifiable parameters in the DBD and EDBD algorithms. As noted, there were three additional parameters (five in all) for the DBD, and eleven parameters in all for the EDBD. It was necessary to reduce the search for suitable parameter settings. In the two-bit XOR problem (the first to be examined) each of the input parameters was varied over a range of values, while all other parameters were held constant. The approximate parameter values resulting in the lowest convergence times over this range were retained for use in testing subsequent parameter settings. In this way the effect of each of the different parameters was noticed, and useful settings were employed. These settings (with some variation) were then used on the other problem types, and on the larger problems. Tables containing the results of parameter modifications on convergence rates (and ratios) are shown and discussed in Chapter 5. The results display the effect of scaling the different parameter values on problems of a fixed size. Subsequent tables contain the results of using similar input parameter values on problems which have been scaled up in size. The effect of adaptive heuristics and their modifiable parameter settings are shown in relation to the results achieved using the standard back-propagation algorithm with different learning rate and momentum settings.

Reporting on the trials was done with respect to overall convergence rates and convergence ratios. Parameter settings were tested in order to achieve a better understanding of their effects on performance as well as to achieve the best results possible. This would make overall comparisons between the algorithms as useful as possible. Although many thousands of trials were made, this corresponds to only a modest sampling of data points across all parameter settings. It is probable that optimum parameter settings were not tested. The results should therefore not be interpreted as an attempt to generate optimum results, but rather as part of a systematic approach to understand and compare different neural network training algorithms.

Due in large part to the unexpectedly large increase in execution times experienced with the scaling up of problem sizes from the small 2 and 3 input cases, a series of timing runs was made to quantify the processing time required. For each of the different problem types and for each size, a separate test was made which initiated a timer after the initial weight files were read and before any calculations were made. On most problem sizes, a total of 100 epochs of processing time was averaged to generate a per epoch number. On larger problems (4-bit Parity, 6 and 11 bit Multiplexer) 10 epochs were used to generate this average time per epoch. These values are contained within the timing comparison tables in Chapter 5.4. They are useful in at least two different ways. A measure of the increase in processing time required as a result of the increase in problem size is readily obtained. Comparison to theoretical rates of increase can be made, and extrapolation of processing time required for larger problem sizes may be made in order to predict training time requirements (for sequential machines). The second use of this data is that it allows another comparison between the effectiveness of the different algorithms to be made. The adaptive algorithms require additional calculations to be made in order to adjust parameters. The resultant increase in the time required may be compared to the overall improvement in convergence rates in order to arrive at a relative measure of the training time required for the different algorithms. It is a measure of performance which has not been considered in most published results, perhaps because it is relevant for sequential processing only. Although this measure is not of much importance for the eventual parallel implementation of learning algorithms, it is useful as long as simulations are made on serial machines.

Chapter 5

Experimental Results

5.1 XOR/Parity Problem

The first problem examined was the 2 bit XOR, with the standard back-propagation algorithm using both learning rate (η) and momentum (α) terms. These parameters were systematically varied over the range $0.0 < \eta \leq 1.0$ and $0.0 \leq \alpha \leq 0.95$. Tests were also run with high learning rates $1.0 < \eta \leq 20.0$. The results are displayed in tabular form in Table 5.1, and displayed graphically in Figure 5.1. The values in Table 5.1 represent the average number of epochs (from 11 runs) required for a 2-2-1 network trained with the standard BP algorithm to learn to correctly classify the four possible inputs to within a total sum of squared error of 0.04. Each data point is the average taken from all converging runs. Runs which did not converge are not included in the average, and the number of non-converging runs is listed as a subscript to each convergence value in the table. The absence of a subscript implies 100% convergence. Figure 5.1 displays the essentially linear decrease in convergence times which result from

		Momentum										
		0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Lrate	0.1	16358	14741	13088	11462	9828	8195	6563	4934	3307	1717	381_7
	0.3	5464	4915	4371	3827	3283	2741	2200	1662	1130	628	222_7
	0.5	3189	2952	2625	2300	1974	1650	1327	1007	696	418	174_7
	0.7	2343	2106	1878	1646	1414	1183	953	727	507	337	149_7
	0.9	1824	1643	1462	1282	1102	923	746	571	404	302	134_7
	1.0	1642	1479	1317	1155	993	832	673	517	368	277	129_7
	1.6	1028	927	826	725	625	526	428	333	250_1	163	101_8
	2.0	824	744	663	583	503	424	347	273_1	215	135_1	89_8
	3.0	552	499	445	392	340	289	240	196	202	98_2	81_7
	4.0	423	378	338	299	260	223	192	165	119	80	65_8
	6.0	209_1	252	322	245	205	186	155	109	76	78	54_7
	8.0	189_1	122_1	108_1	110	108_1	107	93	81_1	61_1	62_3	473_{10}
	10.0	149_5	180_4	243_4	100_1	83_1	172_2	72_1	48_4	54_8	45_9	—$_{11}$
	12.0	185_5	147_5	119_4	192_5	111_6	188_5	168_9	92_8	64_{10}	—$_{11}$	—$_{11}$
	14.0	215_5	163_5	213_5	190_4	164_6	326_8	44_{10}	—$_{11}$	109_{10}	—$_{11}$	—$_{11}$
	16.0	221_5	266_5	174_7	—$_{11}$	260_{10}	156_8	—$_{11}$	125_{10}	—$_{11}$	—$_{11}$	—$_{11}$
	18.0	227_6	289_8	112_9	115_{10}	292_{10}	40_{10}	61_{10}	197_{10}	—$_{11}$	—$_{11}$	—$_{11}$
	20.0	186_6	129_9	64_{10}	56_{10}	140_9	127_8	104_8	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$

Table 5.1: η vs α Convergence Times (in Epochs) for the 2-bit XOR Problem Using the BP Algorithm.

increases in either the learning rate or the momentum term. For the momentum parameter, the linear reduction in convergence times extends from 0.0 through approximately 0.9. The rate of 0.95 increases the improvement, but occasionally at a reduced rate. Combined with larger step sizes, momentum values of .9 and .95 also led to increased non-convergence, measured as a ratio of failed runs to the total attempted. Momentum set at 1.0 led to very poor results, and was not tested on any other problems. Non-convergence resulting from either large step sizes or large momentum values appear to be caused by weight adjustments which are too large to allow the algorithm to effectively follow the error gradient, causing solutions to be stepped over, rather than being reached. Learning rates greater than 1.0 (Table 5.1) were very useful in lowering convergence times. It was not until rates of 8.0 and 10.0 (and greater) were used that large numbers of trials ending in non-convergence began to be noticed. Non-convergence tends to

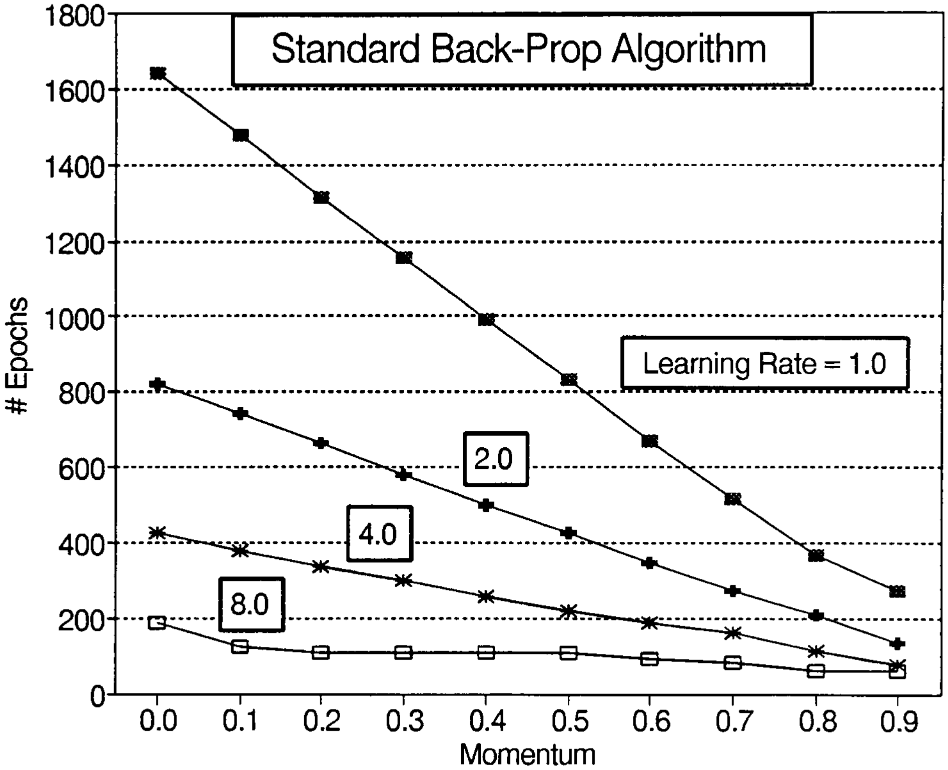


Figure 5.1: Graph of Convergence Times for Selected η Values for the 2-bit XOR Problem Using the BP Algorithm

increase with larger learning rates and momentum terms. As shown in this table, there is a large region in the (η, α) parameter space which produces linearly decreasing convergence times as either learning rate or momentum are increased. It is noted that essentially two orders of magnitude of improvement in convergence times (16385 vs. 277) were realized by modifying the (η, α) parameters from (0.1, 0.0) to (1.0, 0.9). A further improvement was made by further increasing the learning rate beyond 1.0. The best result generated (with all runs converging) was 76 epochs with parameter settings of (6.0, 0.8). The results in [Jacobs 88], which describe the reduction in convergence times with DBD, apparently use convergence data from settings of (0.1, 0.0) for BP as the basis for comparing the

		Update Coefficient					
		0.1	0.2	0.3	0.4	0.5	0.6
Lrate	0.1	185_1	156_1	137_2	139_3	165	186_4
	0.2	181_2	150_1	143_4	137_4	174_4	177_7
	0.3	181_2	144_2	128_3	136_5	153_7	170_8
	0.4	202	145_1	137_5	140_7	172_6	165_{10}
	0.5	172_1	144_3	145_8	151_8	174_{10}	---_{11}
	0.6	192	151_5	136_7	133_8	---_{11}	---_{11}
	0.7	176_1	145_5	143_9	134_9	---_{11}	---_{11}
	0.8	180_2	141_6	---_{11}	---_{11}	---_{11}	---_{11}

Table 5.2: Convergence Times with Varying Update Coefficient for the 2-Bit XOR Problem Using the Gradient Reuse Algorithm.

results of DBD. With the results of Table 5.1, it is apparent that a better basis for comparison is with higher rates of both learning rates and momentum. Although reduced convergence times were noted with continued increases in the learning rate and momentum parameters, an increase in the number of runs which did not converge was also seen. Table 5.1 provides a baseline which may be used to compare the effectiveness of other algorithms against the standard BP algorithm in learning the 2-bit XOR problem.

The Gradient Reuse Algorithm was implemented next and tested against the same problem. As in Hush and Salas [Hush 88], the low and high limits on the learning rate update were set at 5 and 10 gradient reuses. Less than 5 reuses resulted in a reduction of the learning rate by the specified increment. Ten reuses resulted in an increase by the increment. Gradient reuse counts between 5 and 10 updated the learning rate proportionately. The update increment was initially set at 0.1 and allowed to vary in 0.1 increments for various learning rates. The results are summarized in Table 5.2. This table also contains the average number of epochs (averaged over 11 runs) required to successfully

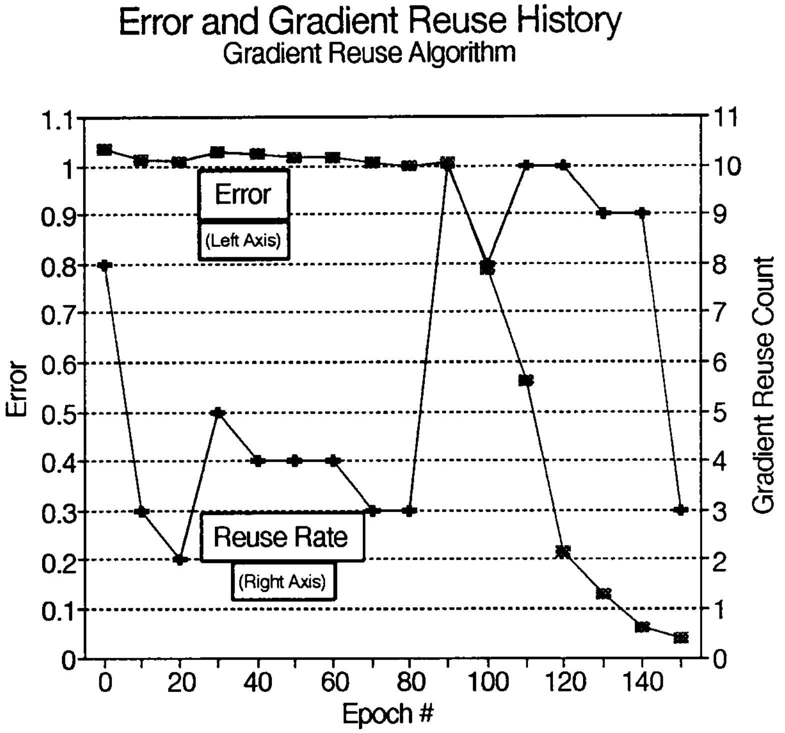


Figure 5.2: Overall System Error and Gradient Reuse Rates for the 2-Bit XOR Problem Using the GRA.

train the network to within an error tolerance of 0.04 total sum of squared error using the parameters specified. It should be noted that the best average result was achieved with an update increment set to 0.5, although this occurs in a region dominated by large numbers of non-converging runs. An increment level of 0.1 reduces the number of non-converging runs, while increasing overall convergence times modestly. The value of 0.1 was used in subsequent testing. Figure 5.2 displays the results of a run for the GRA. It shows the initial slow reduction in overall error, and the concurrent small gradient reuse factor. As relative weight adjustments are made which correctly align weights with respect to one another, continued reapplication of the gradient (as indicated by the high reuse

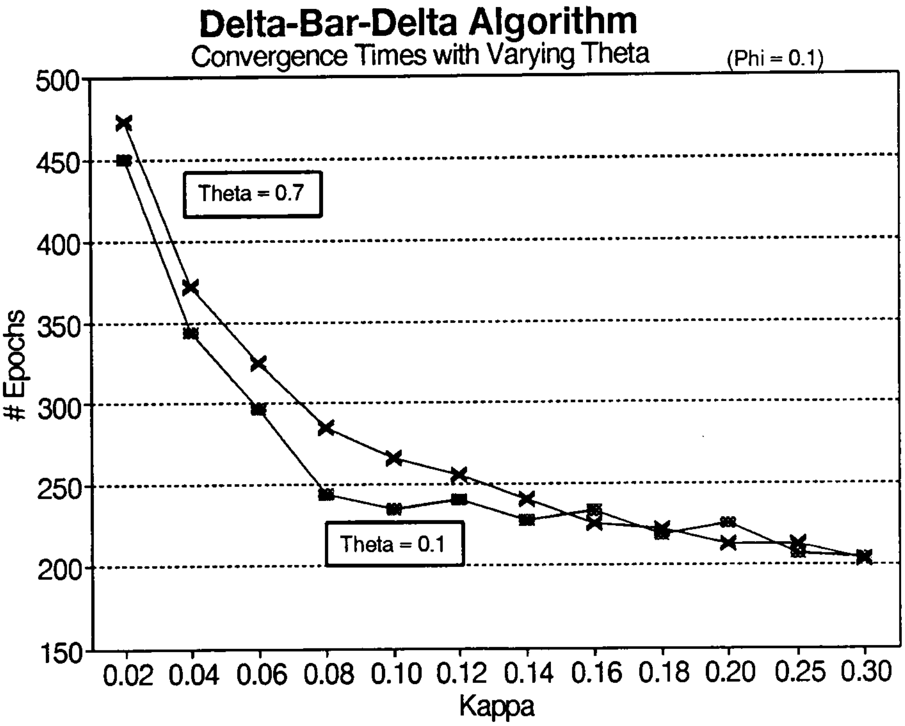


Figure 5.3: Graph of Convergence Times with Varying κ and θ Parameters, using the DBD Algorithm on the 2-Bit XOR Problem.

rates) results in higher learning rates and lower overall error. The algorithm then converges rapidly from an error level of approximately 1.0 to 0.04.

The Delta Bar Delta algorithm was first tested against a range of settings for κ, ϕ, and θ. Each was varied while holding the other terms constant. The dominant term was κ, which determines the amount of increase for the learning rate parameter. As shown in Figure 5.3, convergence times were reduced by 50% when increasing κ from 0.02 to 0.30. It should also be noted that little difference in convergence times resulted from changes in θ over the interval of 0.1 to 0.7. Results were similar for values of ϕ varying from 0.1 to 0.4. Increasing κ beyond 0.2 resulted in slightly lower convergence times, but caused a significant increase

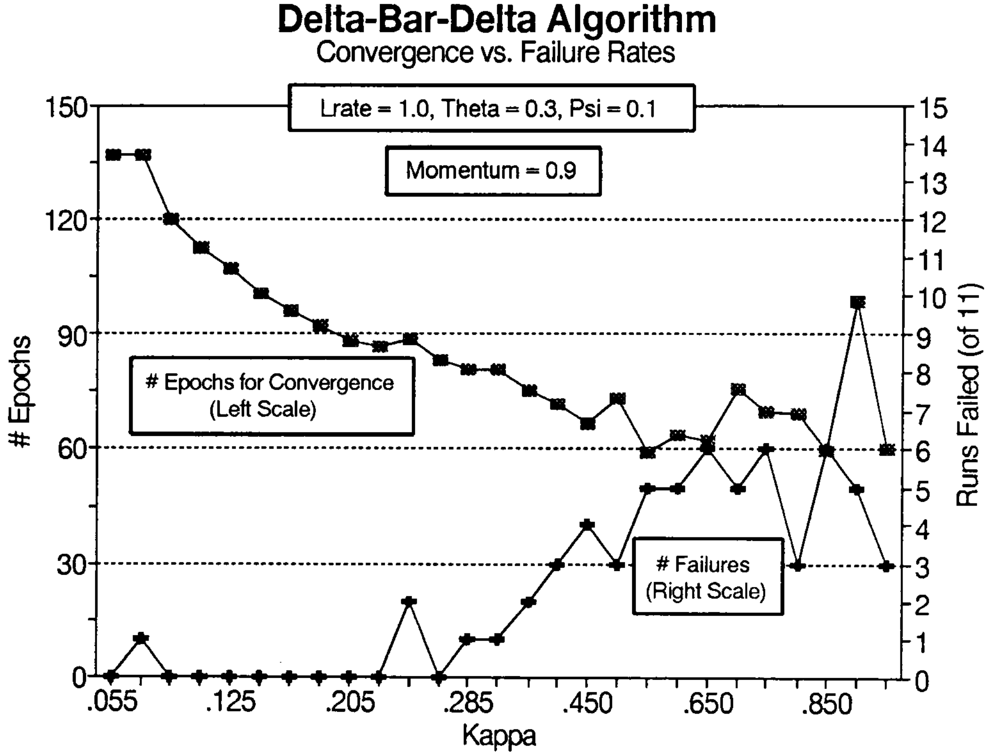


Figure 5.4: Convergence Time vs. Failure Rates with Varying κ for the 2-Bit XOR Problem Using the DBD Algorithm.

in the number of non-converging runs, as is shown in Figure 5.4. κ was usually set to a value in the range of 0.05 to 0.3, in order to maximize the number of converging runs. With the adaptive learning rate parameter, the convergence times did not differ by much when using an initital rate in the range of $0.1 < \eta \leq 1.0$, as shown in Figure 5.5.The adaptive capability enabled the algorithm to make modest adjustments quickly, which usually compensated for small differences in the initial learning rate settings. The initial value of the momentum term, however, is still very important. As with the standard BP algorithm, increases in momentum up to 0.9 generally produced reductions in convergence times. Figure 5.5 also shows this improvement with higher momentum values. This trend was

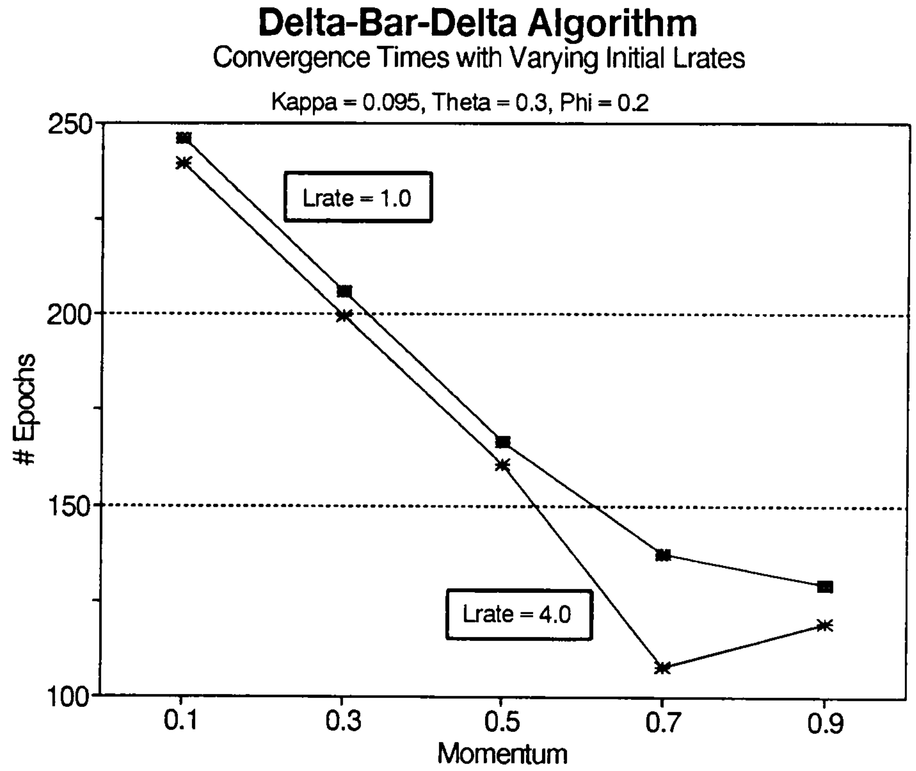


Figure 5.5: Convergence Times with Varying α and Initial η for the 2-Bit XOR Problem Using the DBD Algorithm.

seen across all parameter settings. The best results for both the BP and the DBD algorithms occurred with large momentum values. When large κ values were used, particularly with small ϕ, learning rates often became too large too quickly. These large rates often caused network weight values to become too large in absolute value, which in turn forced (in the backward weight update pass) the derivative of the sigmoid to be evaluated at extreme points resulting in near-zero weight updates. The network thus becomes effectively 'paralyzed' [Wasserman 89]. To prevent this from occurring, an upper bound for η was implemented, similar to the specification for the EDBD. This upper bound became an additional parameter specified at execution time. Values from 1.0 to 40.0 were tested, with

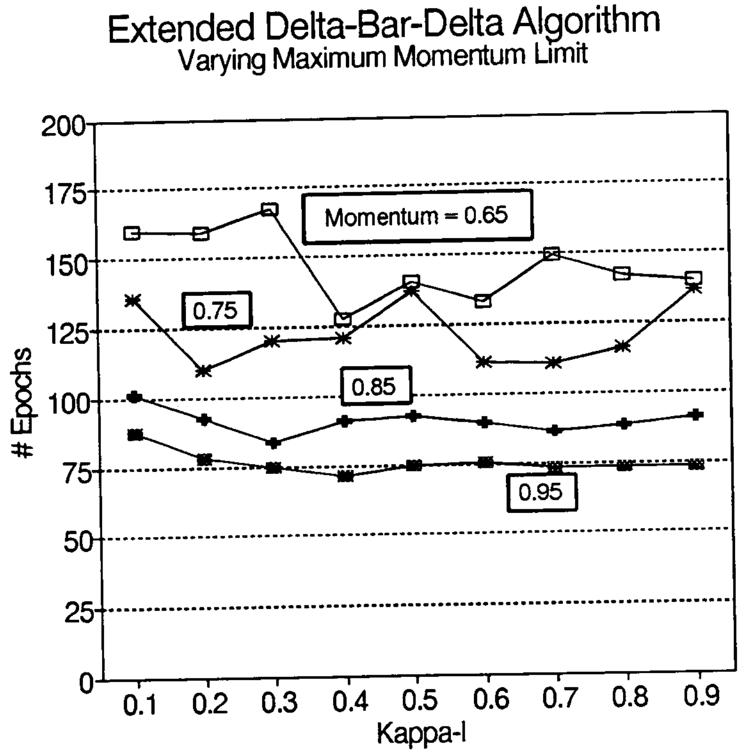


Figure 5.6: Effect on Convergence Time with Varying κ_l and Maximum Momentum Limits Using the EDBD Algorithm on the 2-Bit XOR Problem.

an overall maximum of 5.0 to 30.0 producing the best convergence times with fewer non-converging runs.

The Extended Delta Bar Delta algorithm was tested in much the same way, with each of the parameters allowed to vary while all others were held fixed. Values which produced lower convergence times were utilized in subsequent runs. The effect of parameter settings in the EDBD algorithm were very similar to those seen in the DBD algorithm. As shown with the DBD algorithm, increasing values of κ_l tended to reduce convergence times. At some point, however, this began to cause a greater number of non-converging runs. As with the DBD testing, values in the range of $0.01 \leq \kappa_l \leq 0.4$ generated good convergence times with a

Momentum

		0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
	0.1	4750_1	4208_1	3446_1	3328_1	2892_1	2414_1	1937_1	2846_1	998_1	552_1	388_1
	0.3	1600_1	1447_1	1288_1	1129_1	971_1	813_1	657_1	557	358	236	162
	0.5	1056_1	917_1	799_1	709_1	610_1	494_1	403_1	317	230_1	152	117_3
	0.7	675_1	608_1	547_1	485_1	423_1	357_1	302	234	172_1	113	107_3
	0.9	556_1	505_1	447_1	389_1	335_1	303	234	185_1	144	96	101_1
	1.0	506_1	452_1	406_1	350_1	303_1	265	212	170	137_1	90	92_1
Lrate	1.6	307_1	279_1	265	227	194	168	143_1	89	87	73_1	83_4
	2.0	263	225	203	183	158_1	138_1	120	84	73	59_1	83_6
	3.0	140_1	131_1	129	136	137_1	106_1	83_1	63_2	56_3	88_4	42_7
	4.0	171_2	128	93_2	83_1	74_1	58_1	54_2	54_1	57_2	57_9	45_9
	6.0	244_4	187_3	189_4	155_5	167_5	144_9	154_7	117_{10}	$—_{11}$	$—_{11}$	$—_{11}$
	8.0	347_4	272_4	292_5	305_8	260_{10}	$—_{11}$	90_{10}	181_{10}	$—_{11}$	$—_{11}$	$—_{11}$
	10.0	206_8	126_8	$—_{11}$	$—_{11}$	$—_{11}$	$—_{11}$	$—_{11}$	$—_{11}$	$—_{11}$	$—_{11}$	$—_{11}$

Table 5.3: Convergence Times with Varying η vs. α for the 2-Bit XOR Problem with Sigmoid Coefficient Set to 1.5.

minimum of non-converging runs. Settings for ϕ_l and θ, while less critical than κ_l, were also found to be most useful in the same ranges found in the DBD testing. These settings tended to be small, but non-zero. Values of 0.1 thru 0.4 were used for most tests. κ_m, the parameter which directly affects the momentum term, was more difficult to initialize to suitable values than κ_l. Unlike the learning rate parameter, the momentum term could not be set to exceed 1.0. Hence, any adjustments to the momentum term would have to be small. Values for κ_m were therefore kept in the range of 0.01 to 0.05. Values of 0.1 worked, but seemed to drive the momentum term to its maximum very quickly, which tends to reduce the impact of a tunable parameter. Smaller update values were usually used. The effect of the maximum level for momentum is shown in Figure 5.6. Over a wide range of κ_l settings, improved performance is achieved with increasing maximum momentum settings, up to a level of 0.95. Coupled with the small update increment, κ_m, was the decrement parameter, ϕ_m. It was found to work best with small non-zero settings in the range of 0.1 to 0.3. The two γ parameters

Momentum

Lrate	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
0.1	2123_2	1912_2	1701_2	1490_2	1279_2	1069_2	860_2	655_2	453_3	288_1	247_1
0.3	674_3	640_2	570_2	501_2	432_2	364_2	307_2	234_3	142_3	127_1	168_2
0.5	428_2	386_2	345_2	305_2	264_2	234_2	208_2	156_2	117_3	85_2	114_2
0.7	307_2	278_2	249_2	221_2	204_2	166_3	137_3	111_3	97_2	84_2	113_3
0.9	241_2	219_2	205_2	178_3	154_2	131_3	110_2	94_3	73_2	64_2	58_3
1.0	232_2	211_2	182_2	160_3	139_2	160_3	101_2	91_3	69_2	67_2	56_3
1.6	131_2	124_2	126_3	130_3	125_3	81_3	70_3	64_3	52_2	91_3	32_4
2.0	129_1	105	83_1	80_4	74	68_1	66_1	53_3	70_5	66_6	40_7
3.0	166_1	131_3	134_2	100_1	142_2	70_5	94_4	88_5	99_9	23_{10}	21_{10}
4.0	233_4	233_4	166_3	120_3	86_8	99_6	73_8	124_8	68_{10}	167_{10}	—$_{11}$
6.0	139_7	177_7	106_{10}	—$_{11}$	52_{10}	116_{10}	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$
8.0	312_{10}	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$
10.0	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$

Table 5.4: Convergence Times with Varying η vs. α for the 2-Bit XOR Problem with Sigmoid Coefficient Set to 2.0.

were found to have very little overall impact on convergence times. Values of 0.1 through 10.0 were used with little difference in resulting convergence times noted. These parameters were typically set to 0.1.

The modification to the sigmoid 'sharpness' coefficient resulted in lower convergence times. It also caused larger numbers of non-converging runs. Results from using a sigmoid coefficient of 1.5 are shown in Table 5.3. Coefficient settings of 2.0 (Table 5.4) resulted in a higher number of non-converging runs across a broad range of learning rate and momentum settings. A value of 1.5 improved convergence times of the converged runs substantially while allowing only a modest increase in the number of non-converging runs, as compared to the BP algorithm, shown in Table 5.1. Smaller coefficients (< 1.5) resulted in reduced convergence times (compared to BP), while still maintaining 100% convergence across a fairly broad range of η and α values. Results for coefficient settings of 1.5 and 2.0 are graphed against BP and pre-set weights (all with $\eta = 1.0$) in Figure 5.7.

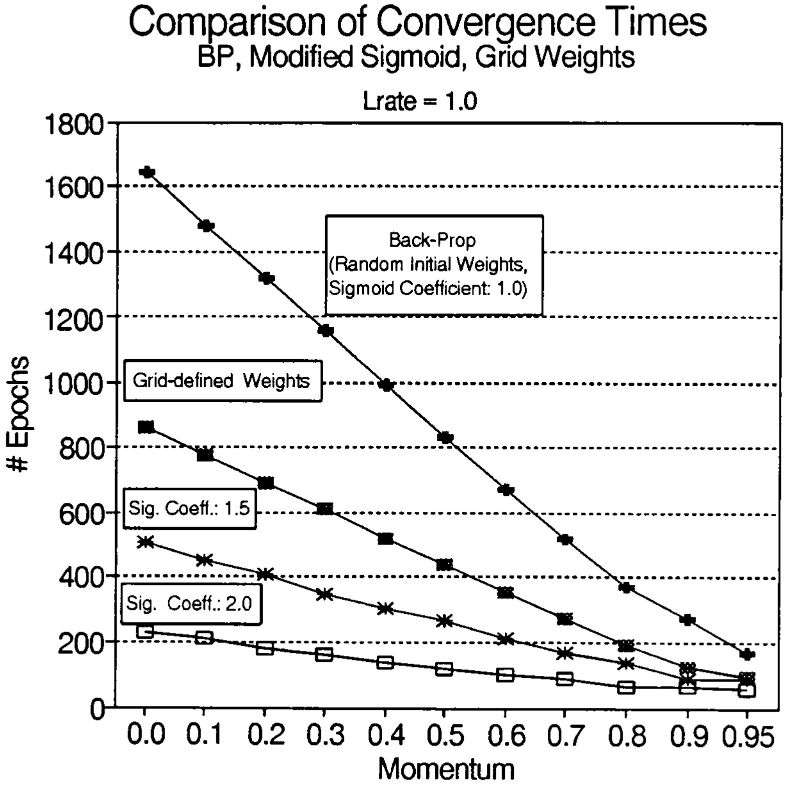


Figure 5.7: Graph of Convergence Times using Random and Predefined Initial Weights, and Modified Sigmoid Coefficients with the BP Algorithm on the 2-Bit XOR Problem.

It may also be noted that while substantial increases in convergence times are achieved compared to the BP algorithm for low settings of both η and α (74 epochs at (2.0, 0.4) with sigmoid coefficient set to 2.0 versus 503 epochs with the coefficient at 1.0) the use of a coefficient $>$ 1.0 can cause a large number of non-converging runs at settings of η and α that had produced low convergence times for the standard BP algorithm. The best result from Table 5.8 for sets of trials with 100% convergence was 74 epochs, which occurred at (2.0, 0.5) with a sigmoid coefficient of 2.0, and was 73 at (2.0, 0.8) with a coefficient of 1.5 (Table 5.3. These compare favorably with the best result achieved with BP of

Momentum

Lrate

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
0.1	8612	7756	6895	6034	5174	4315	3455	2598	1744	903	503
0.3	2875	2588	2301	2015	1729	1443	1140	875	595	324	212_1
0.5	1726	1554	1383	1211	1040	869	699	530	365	208	153
0.7	1234	1111	989	867	745	623	502	383	266	162	119
0.9	960	865	770	675	581	486	393	300	211	214_1	101
1.0	865	779	693	608	522	438	354	272	191	122_1	95
1.6	544	488	435	382	329	277	228	174	126	88	74
2.0	434	391	349	307	265	223	182	142	105	74	61_1
3.0	291	260	234	206	179	151	125	99	99_1	76	49
4.0	218	198	177	156	136	116	96	81	61	58	43
6.0	132	152_1	131_1	169	111	134	69	53	43	40	64_1
8.0	86	75	69	67	112_1	65_1	54_1	83_1	45_2	266_3	36_6

Table 5.5: Convergence Times with Varying η vs. α for the 2-Bit XOR Problem Using Predefined Initial Weights.

76.3 epochs at (6.0, 0.8).

The use of predetermined weights for hidden layer nodes resulted in an overall reduction in convergence times of approximately 50% over a wide range of learning rate and momentum values as compared to BP results. Apparently, the three mutually orthogonal hyperplanes which are produced with the use of the pre-specified weight settings allow the network to more quickly develop the relationships between inputs and their respective desired outputs. The hyperplane boundaries are adjusted in order to classify the inputs with reduced error. The initial specification of the boundaries reduces the number of epochs required to make these adjustments. The convergence times are shown in Table 5.5, and can be compared directly to the BP results in Table 5.1. The use of the predefined weights to specify separating hyperplanes in input pattern space was also used on the DBD and the EDBD algorithms with similar results. Figure 5.8 shows the effect on convergence times of the two different initial weight cases (random and prespecified), for different values of ϕ_l with the EDBD algorithm.

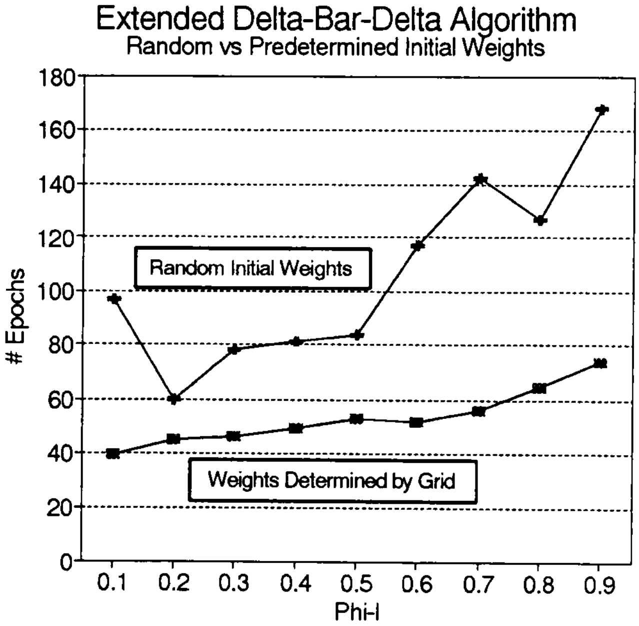


Figure 5.8: Graph of Convergence Times using Random and Predefined Initial Weights wich the EDBD Algorithm on the 2-Bit XOR Problem.

After the generation of results with the 2-bit XOR Problem, the 3-bit and 4-bit parity cases were implemented and tested. The results for the BP algorithm are shown in Table 5.6. A similar response to the 2-bit case is seen for both, with several noticable distinctions. For the 3-bit problem, the number of epochs for convergence has actually dropped from comparable settings with the 2-bit problem (2065 vs 2300 at (0.5, 0.3)). The upper limit on useful settings of the learning rate coefficient (those which generate 100% convergence) has also been reduced to approximately 3.0 (from 6.0 to 8.0) with the 2-bit problem. The 4-bit parity problem produced results which were much different. The number of epochs required to converge increased sharply across all parameter settings, and non-convergence became much more widespread, even occurring at settings of (0.7, 0.3). The effect of using sigmoid coefficients of 2.0 and 1.5, as was done in

Size	Algorithm	Lrate	Momentum				
		(η)	0.1	0.3	0.5	0.7	0.9
3 Bit	**Back-Prop**	0.5	2642	2065	1448	836	328
		1.0	1631	1184	793	428	206
		2.0	861	645	438	223	217
		3.0	673	456	216	163_1	124_2
	Sig: 1.1	0.5	2238	1672	1142	615	255
		1.0	1344	981	652	327	199
		2.0	787	587	395	169	77_3
	Grid	0.5	2593	1803	1247_1	641_1	279
		1.0	1682	1391	742	324_1	151
		2.0	1343	853	316	203	115_1
4-Bit	**Back-Prop**	0.5	8035_1	7677	7772	6405	1682_1
		0.7	6891	4558_1	8049	17668_2	1904_2
		0.9	5056	4083	3045	1974_1	5139_3
	Grid	0.5	7790_4	7542_3	4801_3	2643_4	1176_2
	Sig: 1.1	0.5	$—_5$	$—_5$	2882_4	3189_1	1193_3

Table 5.6: Convergence Times with Selected Parameter Settings for the 3 and 4 Bit XOR/Parity Problems Using the BP, Grid, and Sigmoid Modification Algorithms

the 2-bit problem, resulted in essentially no runs converging. The use of a value of 1.3 for the 3-bit case resulted in an improvement over the BP algorithm, however for the 4-bit case, only non-convergence resulted. The use of prespecified weights (to form a 'grid') yielded an overall improvement in performance in the 3-bit case. The DBD and EDBD adaptive algorithms had similar results for learning the 3 and 4 bit problems. Table 5.7 shows characteristic convergence times for different parameter settings while κ is varied. Both algorithms were able to learn the 3-bit case with reasonably normal settings for their parameter values. The 4-bit case was much different. Most parameter settings resulted in high numbers of non-converging runs. As was seen with the 4-bit BP convergence table, many sets of trials contained one or more non-converging runs. The possible cause of this inability to scale is discussed in a subsequent paragraph. Although all algorithms had trouble scaling up to the 4-bit problem size, the adaptive algorithms did achieve better (lower) convergence times than the BP algorithm (Table 5.8). The

Size	Algorithm	Parameter Settings									
		η	α	ϕ	θ	κ_l					
						0.05	0.1	0.2	0.3	0.4	0.5
3-Bit	**DBD**	1.0	.1	.25	.45	312	261	214	208	198	184
		1.0	.1	.25	.15	326	267	234	266	220	306
		1.0	.2	.25	.15	291	241	216	195	184	198
		2.0	.2	.25	.15	302_1	232	206	194	210	195
	Grid	1.0	.1	.25	.15	442	297	303	274	187	176
		1.0	.2	.25	.15	417	291	185	169	163	195
		2.0	.2	.25	.15	317	268	175	160	250	181
	EDBD	1.0	.1	.15	.05	120	156	236	213	239	175
		1.0	.1	.25	.05	95	112	150	172	187	173
		1.0	.1	.35	.05	81	87	154	176	192	180
		2.0	.2	.25	.05	98	93	88	88	91	109
		4.0	.2	.25	.05	105	85	284_1	323_1	271_1	250_1
		6.0	.2	.25	.05	272_1	273_2	283_2	275_2	271_2	381_1
	Grid	1.0	.1	.15	.1	122	96	81	69	63	61
		2.0	.2	.15	.1	67	66	69	70	74	80
		4.0	.2	.25	.05	73	63	143_1	138	234	292_1
4-Bit	**DBD**	1.0	.9	.1	.1	836_2	715_1	802_3	842_3	812_4	$—_5$
		0.4	.4	.3	.1	912_2	813_2	807_1	742	788_1	
	EDBD	2.0	.5	.3	.05	785_3	933_3	603_2	673_1	856_2	$—_5$
		1.0	.1	.35	.05	398_4	250_3	480_3	730_4	$—_5$	$—_5$

Table 5.7: Convergence Times with Selected Parameter Settings for the 3 and 4 Bit XOR/Parity Problems Using the DBD and EDBD Algorithms

best results achieved from each algorithm on each of the problem sizes are listed here. The difference in performance increased with the small increases in problem size. The standard BP algorithm performed comparatively well with the 2 and 3 input problems, but showed poor convergence rates for the 4 input problem. The Gradient Reuse Algorithm performed less well. Although it produced a slightly better average rate of convergence for the 2 input problem, the best results with the 3 input problem were achieved with only 4 of 5 runs converging. This was the best response achieved over a reasonable range of parameter settings. In no case did 5 of 5 runs converge. This result was repeated in the 4 input problem. In both cases, convergence times were worse than with the standard BP algorithm. The use of the predefined weights and the sigmoid sharpness modification both were of benefit in reducing average convergence times. Their

Size	Algorithm	#	Mean	Std Dev	Min	Max	η	α	Other Parameters
2-Bit	Std. Back-Prop.	11	181.8	54.8	110	284	0.9	0.95	
	High Lrate		76.4	26.2	46	135	6.0	0.8	(for Lrates > 1.0)
	Mod. Sig.		73.0	18.2	51	104	2.0	0.8	Sig. Coeff: 1.5
	With Grid		40.0	11.3	28	69	6.0	0.9	
	Grid, Sig.		36.6	6.0	26	46	6.0	0.9	Sig. Coeff: 1.25
	Gradient Reuse		165.0	34.5	88	219	0.1	0.0	Upd. Coeff: 0.1
	Delta-Bar-Delta		83.2	30.9	51	168	1.0	0.9	κ: .256, ϕ: 0.1, θ: 0.3
	With Grid		74.6	24.7	51	147	0.5	0.9	κ: .256, ϕ: 0.1, θ: 0.3
	Extended DBD		43.5	8.6	33	66	10.0	0.2	κ_l: .256, ϕ_l: 0.1, γ_l: 0.5, θ: 0.3
	With Grid		31.7	9.9	23	51	14.0	0.2	κ_m: .08, ϕ_m: .05, γ_m: 0.1
3-Bit	Std. Back-Prop.	11	206.7	110.0	107	519	2.0	0.85	
	Mod. Sig.		133.6	50.7	73	217	1.5	0.75	Sig. Coeff: 1.3
	With Grid		104.2	18.2	85	132	3.0	0.75	
	Grid, Sig.		95.2	24.8	64	125	2.0	0.7	Sig. Coeff: 1.1
	Gradient Reuse	5	1387.2	917.2	388	2384	0.1	0.0	Upd. Coeff: 0.1 (4/5 Runs)
	Delta-Bar-Delta	11	172.8	37.9	119	248	1.0	0.2	κ: .7, ϕ: 0.25, θ: 0.15
	With Grid		141.3	22.4	110	173	1.0	0.2	κ: .6, ϕ: 0.35, θ: 0.05
	Extended DBD	11	77.6	17.1	53	105	2.0	0.2	κ_l: .2, ϕ_l: 0.35, γ_l: 0.5, θ: 0.05
	With Grid		58.7	9.6	43	74	2.0	0.2	κ_m: .08, ϕ_m: .1, γ_m: 0.1
4-Bit	Std. Back-Prop.	5	3044.6	1121.0	2124	4750	0.9	0.5	
	Gradient Reuse		3894.0	1328.1	2697	5407	0.1	0.0	Upd. Coeff: 0.1 (4/5 Runs)
	Delta-Bar-Delta		638.4	205.1	531	1048	1.5	0.78	κ: .7, ϕ: 0.25, θ: 0.15
	Extended DBD		539.2	144.8	386	776	0.8	0.6	κ_l: .2, ϕ_l: 0.35, γ_l: 0.5, θ: 0.05 κ_m: .08, ϕ_m: .1, γ_m: 0.1

Table 5.8: Summary of Best Average Convergence Times (in Epochs) from Sets of Runs with 100% Convergence, for the XOR/Parity Problem.

beneficial effect was found to hold for the DBD and the EDBD problems, as well. The best result for the BP Algorithm on the 2-bit XOR problem occured using the predefined weights and a sigmoid sharpness coefficient of 1.5. This value was 36.6. The best convergence time for the 2 input problem of 31.7 was achieved by the EDBD algorithm using the grid-defined weights. The best average convergence times for the 3 input parity problem were again achieved with the use of the grid-defined weights and a modified sigmoid coefficient. For the BP algorithm, 95.2 Epochs was the best average number of epochs required to converge to a total sum of squared error level of 0.04. The best average time for the DBD and the EDBD was 141.3 and 58.7, respectively. In each case, the predefined initial weights were used. Figure 5.9 shows graphically the best

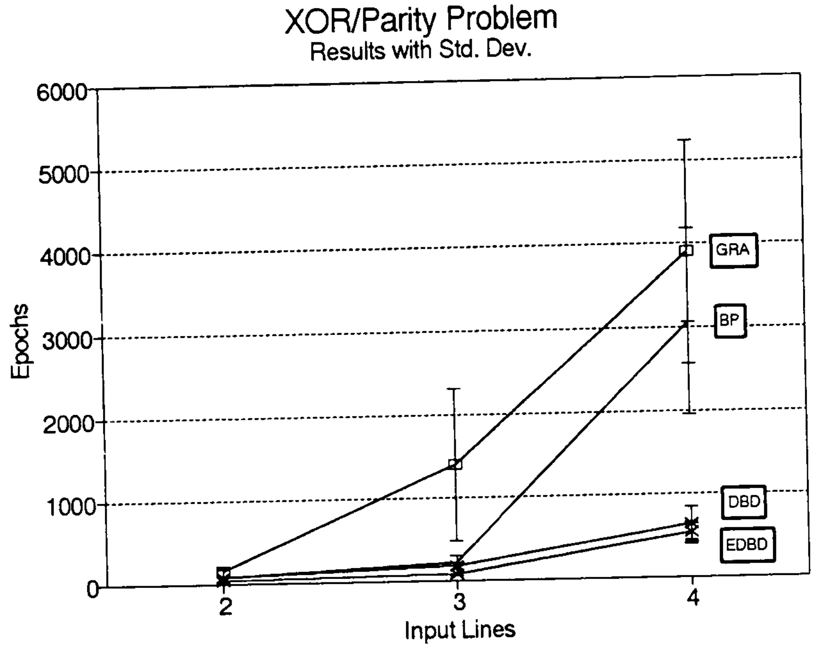


Figure 5.9: Graph of Best Average Convergence Times with Std. Dev. (from Sets of Runs with 100% Convergence) for the Encoder/Decoder Problem.

average convergence times (± one standard deviation) from sets of runs with 100% convergence. No significant difference exists between DBD and EDBD. There is, however a pronounced difference between them and BP and GRA at the 4-bit problem size.

Two problems were encountered when scaling up problem size. First, it was observed that the weight update values were much greater than in the smaller cases. This was particularly true in the 11 bit multiplexer case with 2048 input patterns, where a single weight adjustment was sufficient to prevent further learning. In the standard BP algorithm, the weight adjustment is made by summing per pattern Δweight values over all patterns. While apparently not of importance with the 4 and 8 pattern cases (it might even improve convergence times by producing larger overall weight adjustments), these excessive weight adjustments began showing up as problems in the 16, 64 and 2048 pattern cases. In the 11 bit multiplexer problem (with 2048 patterns), a single epoch was sufficient to force the network into a state from which it never changed. Wasserman [Wasserman 89] attributes this condition, which he termed 'network paralysis', to a process by which large weight adjustments are made which cause a node's activation level (weighted sum of inputs) to be very large (either negative or positive). This impacts the back propagation of error phase in that the derivative of the sigmoid evaluated at this extremely large level is very near zero, causing only very minute weight changes to take place. A network can thus be forced into a state from which it may not recover. This apparently was taking place on these (relatively) large problems. The solution was to normalize the weight adjustment, by taking the average instead of the sum of the weight adjustments. In place of the initial weight update equation:

$$\Delta w_{jk} = \sum_p \Delta_p w_{jk}$$

the following was used:

$$\Delta w_{ij} = \frac{1}{p} \sum_p \Delta_p w_{ij}$$

The second problem noted was the failure of the predefined weight method to produce converging runs in the 4-bit problem, although it had achieved very good comparitive results in the 2, and 3 bit problems. For the 4-bit parity problem, no sets of trials achieved 100% convergence ratios using either the BP, DBD or the EDBD algorithms using the grid-defined weights. It appears that the problem is with the implementation of both the BP algorithm, and the way in which the weight values were assigned. The BP algorithm was set up to accept binary inputs, 0 and 1. The weighted sum of the inputs calculated at each node thus does not reflect any weight on an input line set to 0. Furthermore, on the weight update phase, no weight change is produced for weights on an input line of 0. This limits the rate of learning, since only approximately 50% of input lines are non-zero. Although this may reduce the rate of learning, it certainly does not inhibit it, as reflected in the convergence of the 2 and 3 input problems. The addition of the predefined weights, however, adds a complicating factor: all but one of the weights attached to each hidden layer node is assigned a zero value. Only one weight per hidden layer node is set to a 1. Apparently, the large number of zero values in the network allow virtually no learning to take place. Some additional tests were run with bipolar inputs (-1 and 1) and with the predefined weights set to 0.9 and 0.1 instead of 1.0 and 0.0. This produced better results, with a number of runs converging. However, no set of trials achieved 100% convergence. The implementation could also be sensitive to the random initial settings for weights in the output layer.

In summary, the results of the BP algorithm show decreasing convergence times with increases in η and α. The limits to useful increases in these parameters are also noted. The GRA was found to perform poorly compared to BP. Improvements in convergence times are achieved with modified sigmoid coefficients greater than 1.0. Predefined weights reduce convergence times approximately 50% on the 2 and 3 bit problems, but fail to converge on the 4-bit problem. The DBD algorithm is shown to produce better convergence times than BP on the 3 and 4 bit problems. EDBD achieves significant reductions in convergence times compared to BP on all three problem sizes, but determining the exact settings of parameters to achieve the best results can be difficult.

5.2 Multiplexer Problem Results

The multiplexer problem was examined next. A 3-3-1 network was trained on the 3-bit multiplexer problem using the BP algorithm. The learning rate and momentum terms were allowed to vary over the ranges $0 < \eta \leq 10.0$ and $0 \leq \alpha \leq 0.95$. Increases in both learning rate and momentum coefficients resulted in linearly decreasing convergence times. This was previously observed in the XOR/parity problems, although the number of epochs required to train the network on the multiplexer problem was much smaller: 3181 vs. 16358 at (η,α) of (0.1, 0.0), and 36 vs. 182 at (0.9, 0.95). The region of higher rates of non-convergence was very similar, becoming more pronounced at high rates of both η and α (see Table 5.9) just as with the 2-bit XOR problem.

The use of the pre-defined (or grid-defined) weights did not achieve the same type of results on the multiplexer problem as it did on the XOR. The results for a number of parameter settings are reproduced in Table 5.9, and Table 5.10 for DBD and EDBD algorithms. When compared to results obtained with random initial weights, they are shown to be poorer, although usually only by a modest amount: 173 epochs at (1.0, 0.5) vs. 161 for the standard BP algorithm. This is in contrast to a typical reduction of 50% in the number of epochs required for convergence for the XOR problem. The 3-bit multiplexer problem does not require the creation of 3 separating planes, as does the XOR problem (see Figures 3.1 and 3.2). The performance results indicate that the pre-specification of weights to produce these initial discriminants does not seem to produce a meaningful advantage in learning the multiplexer problem.

The use of sigmoid coefficients greater than 1.0 produces improvements

Algorithm	Lrate	Momentum										
	(η)	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
Back-Prop	0.1	3181	2863	2545	2278	1910	1593	1276	959	642	325	165
	0.3	1061	957	850	744	639	533	427	322	217	108	70.1
	0.5	638	574	511	448	384	321	257	194	131	68.1	50.5
	0.7	456	411	365	320	275	230	185	140	93.5	51.9	41.5
	0.9	355	320	285	250	214	179	144	109	72.6	43.2	36.1
	1.0	320	288	256	225	193	161	130	98.2	65.3	39.8	34.1
	2.0	161	144	129	113	97.2	81.7	66.1	49.6	34.5	25.7	24.1
	4.0	94.9	84.1	72.4	63.5	53.1	44.1	35.5	29.0	25.2	25.3_1	23.4_2
	6.0	63.3	60.3	51.7	48.5	42.3	37.7	29.6	32.4	65.9_1	76.0_3	87.8_5
	8.0	48.7	45.4	40.7	38.0	34.6	34.9	31.2_2	36.0_5	28.0_5	39.5_9	36_1
	10.0	57.7	40.7	40.2_1	49.3	47.3	49.0_2	74.8_5	52_{10}	—$_{11}$	—$_{11}$	—$_{11}$
Grid	1.0	342	308	274	241	207	173	139	105	69.8	39.4	31.8
	2.0	172	155	138	121	105	87.5	70.4	53.0	35.3	24.6	21.5_1
	3.0	123.	108	97.1	82.4	70.7	60.8	47.9	35.2	25.3	19.2_1	17.9_1
Sig: 1.1	1.0	263	236	211	185	159	133	107	80.8	53.6	34.6	30.5
	2.0	134	120	106	93.2	80.1	67.3	54.2	40.5	28.8	22.6	21.5
	3.0	101	87.3	75.3	65.9	56.3	46.9	37.5	28.5	23.0	26.2_1	28.4_2
Sig: 1.2	1.0	230	199	177	155	133	111	89.3	67.5	44.6	30.5	27.9
	2.0	115	102	90.2	78.6	67.5	56.4	45.3	33.7	25.4	21.5	23.1
	3.0	85.1	75.3	65.3	57.1	50.7	40.0	32.2	26.0	22.7	34.0	21.3_3
Sig: 1.3	1.0	187	168	150	131	113	94.5	75.9	56.9	38.3	27.5	25.4
	2.0	101	88.4	78.1	68.3	58.5	48.3	38.6	29.2	23.5	24.0	22.9_1
	3.0	91.1	67.6	65.8	51.3	44.3	38.3	33.3	26.1	33.2	45.8_3	53.7_5
Grid, Sig: 1.2	1.0	220	198	176	154	133	111	89.3	67.6	44.6	30.5	27.9
	2.0	115	102	90.2	78.6	67.5	56.4	45.3	33.7	25.4	21.5	23.1
	3.0	85.1	75.3	65.3	57.1	50.7	40.0	32.2	26.0	22.7	34.0	21.3_3
Gradient Reuse	0.5	93.7	53.8	26.2	17.3_8	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$
	0.7	92.9	53.1	24.2	29.1_1	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$
	0.9	83.6	50.0	24.3	19.3_2	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$
	1.0	87.5	47.5	24.4	24.2_1	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$
	2.0	76.7	47.5	24.4	24.2	28.6_5	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$	—$_{11}$

Table 5.9: 3-Bit Multiplexer Convergence Times Using the Back-Prop, Gradient Reuse, Grid, and Sigmoid Modification Algorithms.

in convergence times which are similar to those achieved with the XOR problem. A small increase in the coefficient (to 1.1 and 1.2) yielded results that were better than with the standard coefficient setting of 1.0. The improvement which was pronounced for non-optimal settings of momentum, became non-existent at high levels of momentum (where the standard BP produced its best results). Settings of the sigmoid coefficient which are too high, coupled with either high levels for η or α may result in runs which do not converge to a solution within the error tolerance. This occurred with a sigmoid coefficient setting of 1.3 with (η,α) settings

Algorithm	Parameter Settings								
	η	α	ϕ	θ	κ_l				
					0.1	0.2	0.3	0.4	0.5
DBD	1.0	.50	.25	.25	58.7	49.7	48.3	41.7	40.8
	1.0	.85	.25	.25	36.4	29.9	27.3	26.5	25.0
	1.0	.85	.25	.45	41.5	34.8	30.5	28.5	28.2
	2.0	.20	.25	.45	74.8	63.5	60.0	57.0	54.0
	2.0	.85	.05	.25	25.8	23.8	23.1	24.0	23.7_2
	2.0	.85	.25	.25	31.9	28.0	26.5	25.0	24.5
	2.0	.85	.25	.45	40.7	31.5	29.0	28.2	27.4
	3.0	.95	.15	.25	23.6	23.5	22.7	22.4_1	23.3_2
	3.0	.95	.25	.25	26.7	23.9	22.4	22.6_1	20.3_1
Grid	2.0	.85	.05	.05	25.5	23.5	22.7	22.4_1	23.3_2
	2.0	.85	.25	.25	31.9	28.0	26.5	25.0	24.5
	2.0	.85	.25	.45	39.0	31.9	29.6	27.9	26.1
EDBD	2.0	0.2	0.15	.05	27.2	25.9	25.7	55.8	109.5
	2.0	0.2	0.25	.05	29.5	26.6	25.6	25.3	26.4
	2.0	0.2	0.35	.05	32.5	28.4	26.7	26.5	26.4
	4.0	0.2	0.15	.05	27.6	25.6	24.5	25.3	26.0
	4.0	0.2	0.25	.05	28.5	26.1	24.3	24.4	24.8
	4.0	0.2	0.35	.05	34.5	28.9	26.6	24.5	24.5
	6.0	0.2	0.15	.05	26.7	24.8	24.0	23.5	23.5
	6.0	0.2	0.25	.05	28.7	25.9	26.2	24.9	24.8
	8.0	0.2	0.25	.15	25.5	23.9	23.3	23.5	23.5
	10.0	0.2	0.25	.15	30.6	31.4	66.5	52.3	54.7
Grid	2.0	.2	.25	.05	26.5	24.5	26.1	27.1	28.7
	2.0	.2	.45	.05	29.8	26.3	27.1	27.1	29.9
	6.0	.2	.25	.05	29.5	30.0	27.4	27.7	26.9

Table 5.10: Convergence Times with Selected Parameter Settings for the 3-Bit Multiplexer Problem Using the DBD and EDBD Algorithms

of (2.0, 0.95), where 1 run failed to converge, whereas all 11 runs had converged using the standard setting of 1.0. Similar results were seen in all problem types. Combining the pre-defined weights with elevated levels of the sigmoid coefficient also did not produce an overall improvement in convergence times. This is shown in Table 5.9 with a number of runs with various parameter settings, and in Table 5.14, which has the best results obtained for each of the algorithms. This table shows that except for the DBD algorithm, the standard BP algorithm performed better than the other algorithms on the 3-bit multiplexer problem. Table 5.10 contains a series of results for the DBD and EDBD algorithms, using a range of input parameters. As shown, the κ (for DBD) and κ_l (for EDBD) can be varied

Algorithm	Lrate (η)	Momentum				
		0.1	0.3	0.5	0.7	0.9
Back-Prop	0.2	516	414	300	188	82
	0.4	476	367	243	164	127_1
	0.6	449	324	200	224_1	—$_5$
	0.8	403	271	123_1	253_2	—$_5$
	1.0	355	168	77_2	112_2	—$_5$
Grid	0.2	659_1	677	487	306	124
	0.6	546	226	170	267	944_2
	1.0	464	528	499_3	—$_5$	—$_5$
Sig: 1.1	0.2	420	339	251	153	77
	0.4	444	283	138	512	208_1
	0.6	464	378	170	267	—$_1$
Grid, Sig: 1.1	0.2	739	553	396	254	348
	0.4	362	236	161	180	343_1
Gradient Reuse	0.1	706_2	645_4	—$_5$	—$_5$	—$_5$
	0.2	568_1	592_2	—$_5$	—$_5$	—$_5$

Table 5.11: Convergence Times with Selected Parameter Settings for the 6-Bit Multiplexer Problem Using the Back-Prop, Gradient Reuse, Grid, and Sigmoid Modification Algorithms.

(achieving good results) over a wider range for the 3-bit multiplexer than was possible for the XOR problem. Levels of 0.4 and 0.5 generally produced the best results, with only a few runs not converging. The DBD algorithm outperformed all others on the 3-bit multiplexer (19.6 vs. 20.9 for the BP algorithm) This small difference may be accounted for in that the multiplexer problem is learned in only 20 epochs, and in this interval, the adaptive parameters in DBD and EDBD are not modified by much. For problems solved in a small number of epochs, the adaptive algorithms perform very similarly to BP.

For the 6-bit multiplexer problem, the best result obtained for the BP algorithm was 81.8 at (0.2, 0.9). The use of a modified sigmoid coefficient (set to 1.1) improved results to 76.8 epochs. The use of grid-defined weights, and the combination of this and a modified sigmoid coefficient performed poorly (124.1 and 148.6, respectively). The Gradient Reuse algorithm performed worst of all.

Algorithm	Parameter Settings								
					κ_l				
	η	α	ϕ	θ	0.05	0.1	0.2	0.3	0.4
DBD	0.1	.1	.1	.1		130.4	96.8_1	98.3_2	$—_5$
	0.1	.2	.1	.1		148.0_1	92.3_2	74.0_4	$—_5$
	0.1	.3	.1	.1		108.0	62.0_4	63.5_3	$—_5$
	0.1	.7	.1	.1		72.0	55.5_1	49.5_3	$—_5$
	0.2	.1	.1	.1		151.2	135.0_3	180.3_2	$—_5$
	0.2	.7	.1	.1		83.2	94.8_1	69.8_1	$—_5$
Grid	0.1	.1	.1	.1		158.7_2	200.0	121.0_4	136.0_3
	0.2	.1	.1	.1		237.2_1	191.2_1	208.3_2	161.0_4
EDBD	0.05	0.1	0.1	.1	61.2	50.5_3	66.0_2	48.0_4	51.0_4
	0.05	0.1	0.2	.1	77.8	68.0	53.2_1	83.7_2	55.5_3
Grid	0.05	0.1	0.1	.1	99.8	75.5_1	98.5_3	68.0_4	162.0_4
	0.05	0.1	0.2	.1	79.2	69.0_1	97.5_1	93.7_2	80.5_3

Table 5.12: Convergence Times with Selected Parameter Settings for the 6-Bit Multiplexer Problem Using the DBD and EDBD Algorithms.

The best result obtained was 568.3 epochs, and this was achieved with only 4 of 5 runs converging. No sets of runs achieved 100% convergence. As shown in Table 5.14, the DBD and the EDBD algorithms performed better than the other types. The DBD achieved its best result of 52.1, and EDBD was next with 61.2 epochs.

The effect of scaling the multiplexer problem up to 6 bits (2 address lines, and 4 data lines) is shown in the results given in Table 5.11 for the BP, its modifications (Grid and Sigmoid Change), and the Gradient Reuse Algorithm. Table 5.12 contains similar run results for the DBD and EDBD. As was noted in the XOR problem, the learning rates which result in the best performance of the network are markedly lower than those used in the 3-bit case. Levels which produced improved results on the 3-bit problem produced large numbers of non-converging runs. This phenomenon was caused by the large number (64) of pattern-level weight update terms which contribute to the weight change at each epoch. The additive impact of 64 of these weight adjustments (8 times the number occurring in the 3-bit case) was such that the learning rate coefficient had to be scaled down by approximately this same factor, in order to achieve convergence.

The initial effect of scaling the multiplexer problem to 11 bits was that no runs ever converged to a solution. The majority of tests would undergo a single weight change which would preclude any further weight changes. As explained in the XOR case, this problem was caused by the weight update resulting from the sum of the 2048 pattern-level Δweight changes. The problem was corrected by modifying the weight update equation to normalize the pattern level Δweight changes. This modification not only corrected the problem of no converging runs, it also allowed use of parameter settings in the same range used for the smaller input cases. Table 5.13 contains results achieved after making the weight adjustment change. For all tested algorithms, good results were noted over a broad range of parameter settings, much as had been noted in the small problem testing. The best result obtained for the BP algorithm was 473.6 Epochs at (6.0, 0.95), which is similar to the best settings found for the 2-bit XOR problem. The DBD and EDBD produced even better results, with only minor problems from non-converging runs. The best results obtained from these algorithms are also contained in in Table 5.14.

Another problem uncovered with the 6 and 11 bit multiplexer problems was with the use of the total sum of the squared error criterion, as described previously. The problem was noted in [Fahlmann 88]. The solution was to replace the 'total sum of squared error criterion' with a 'maximum error per pattern criterion'. This modification was made to the multiplexer problems, and subsequently modified after experience with the encoder/decoder problem (with

Algorithm	Parameters						Epochs
	η	α	κ_l	ϕ_l	θ	κ_m	
Back-Prop	3.0	0.7					2563.0
	3.0	0.9					1712.2
	4.0	0.95					848.4
	6.0	0.95					473.6
Grid	3.0	0.9					1752.0
	6.0	0.95					336.6
DBD	0.2	0.2	0.1	0.1	0.1		3122.2
	1.0	0.9	0.2	0.1	0.1		408.8
	1.5	0.9	0.3	0.1	0.1		336.6
	2.0	0.95	0.35	0.1	0.1		224.8_1
	4.0	0.95	0.35	0.1	0.1		259.6
	6.0	0.95	0.4	0.1	0.1		256.2
Grid	1.5	0.9	0.3	0.1	0.1		361.0
	2.0	0.95	0.35	0.1	0.1		222.2
	4.0	0.95	0.40	0.1	0.1		247.4
	6.0	0.95	0.40	0.1	0.1		201.6
EDBD	0.2	0.2	0.05	0.1	0.1	0.1	680.0
	0.4	0.4	0.3	0.1	0.1	0.05	285.4
	0.5	0.5	0.3	0.1	0.1	0.07	401.0
	0.6	0.6	0.3	0.1	0.1	0.03	242.4
	1.5	0.9	0.4	0.1	0.1	0.05	260.4
	2.0	0.95	0.4	0.1	0.1	0.02	343.8
	4.0	0.95	0.4	0.1	0.1	0.02	302.2
Grid	0.6	0.6	0.3	0.1	0.1	0.03	216.0_1
	1.5	0.9	0.4	0.1	0.1	0.05	329.8
	2.0	0.95	0.4	0.1	0.1	0.02	352.6
	4.0	0.95	0.4	0.1	0.1	0.02	522.7_2

Table 5.13: Convergence Times with Selected Parameter Settings for the 11-Bit Multiplexer Problem Using the Back-Prop, DBD and EDBD Algorithms with Normalized Weight Update

multiple output nodes). It is described in the following section.

The reduced complexity of the multiplexer problem is readily apparent when comparing the 'best' results for the 3-bit XOR and the 3-bit Multiplexer trials found in Tables 5.7 and 5.14. The reduction in epochs needed to converge ranged from 50% for the EDBD, 80% for the DBD, to almost 90% for the BP algorithm. In this set of trials, the use of the predefined weights did not produce the 50% reduction in convergence times that was noted with the small XOR problem. The use of an increased sigmoid sharpness coefficient was not found to be as effective as with the XOR problem. Only the BP algorithm

Size	Algorithm	#	Mean	Std Dev	Min	Max	η	α	Other Parameters
3-Bit	Std. Back-Prop.	11	20.9	2.4	18	26	3.0	0.95	
	Mod. Sig.		21.5	2.4	19	26	2.0	0.95	Sig. Coeff: 1.1
	With Grid		24.6	3.1	21	30	2.0	0.95	
	Grid, Sig.		21.9	3.1	18	28	2.0	0.85	Sig. Coeff: 1.2
	Gradient Reuse		24.2	3.5	19	33	0.7	0.2	Upd. Coeff: 0.1
	Delta-Bar-Delta		19.6	3.5	15	28	3.0	0.95	κ: .5, ϕ: 0.15, θ: 0.25
	With Grid		23.7	4.1	18	33	10.0	0.2	κ: .5, ϕ: 0.15, θ: 0.25
	Extended DBD		23.3	4.1	18	33	10.0	0.2	κ_l: .3, ϕ_l: 0.25, γ_l: 0.5, θ: 0.15
	With Grid		24.6	3.6	20	34	2.0	0.2	κ_m: .08, ϕ_m: .05, γ_m: 0.1
6-Bit	Std. Back-Prop.	11	81.8	10.7	67	95	0.2	0.9	
	Mod. Sig.		76.8	10.7	67	95	0.2	0.9	Sig. Coeff: 1.1
	With Grid		124.1	55.3	64	220	0.2	0.9	
	Grid, Sig.		148.6	78.1	138	205	0.4	0.5	Sig. Coeff: 1.2
	Gradient Reuse		568.3	262.8	326	812	0.2	0.1	Upd. Coeff: 0.1
	Delta-Bar-Delta		52.1	6.3	44	61	0.1	0.9	κ: .7, ϕ: 0.1, θ: 0.1
	With Grid		70.8	18.2	53	98	0.1	0.1	κ: .7, ϕ: 0.1, θ: 0.1
	Extended DBD		61.2	7.1	55	75	0.1	0.1	κ_l: .05, ϕ_l: 0.1, γ_l: 0.1, θ: 0.1
	With Grid		79.2	18.5	61	113	0.1	0.1	κ_m: .08, ϕ_m: .05, γ_m: 0.1
11-Bit	Std. Back-Prop.	5	473.6	143.2	343	750	6.0	0.95	
	With Grid		473.4	143.1	343	750	6.0	0.95	
	Gradient Reuse		—	—	—	—	—	—	No Runs Converging
	Delta-Bar-Delta		256.2	71.9	169	350	6.0	0.95	κ: .4, ϕ: 0.1, θ: 0.1, Max η: 40
	With Grid		201.6	25.6	163	231	6.0	0.95	κ: .4, ϕ: 0.1, θ: 0.1, Max η: 40
	Extended DBD		408.4	53.1	309	455	0.6	0.6	κ_l: .1, ϕ_l: 0.1, γ_l: 0.1, θ: 0.1
	With Grid		329.8	90.5	191	456	1.5	0.9	κ_m: .08, ϕ_m: .05, γ_m: 0.1

Table 5.14: Summary of Best Average Convergence Times (in Epochs) from Sets of Runs with 100% Convergence, for the Multiplexer Problem.

operating on the 6-bit Multiplexer problem benefited from its use, and the improvement in convergence times was only 76.8 vs 81.8. For large problem sizes the GRA performed poorly. In the 3 input case, it required approximately the same number of epochs as BP to reach the same 0.04 error level. In the 6 input case, however, it required six times the number of epochs as did the BP, and more than ten times the number of epochs required by the DBD algorithm. In the 11-bit multiplexer problem, no runs converged for the GRA. All trials which were run became trapped by local minima. A number of initial learning rate and update coefficient settings were attempted without success. It is not clear why the GRA fails to converge on the largest problem sizes, however, increasing numbers of non-converging runs were noted as problem size was increased. For

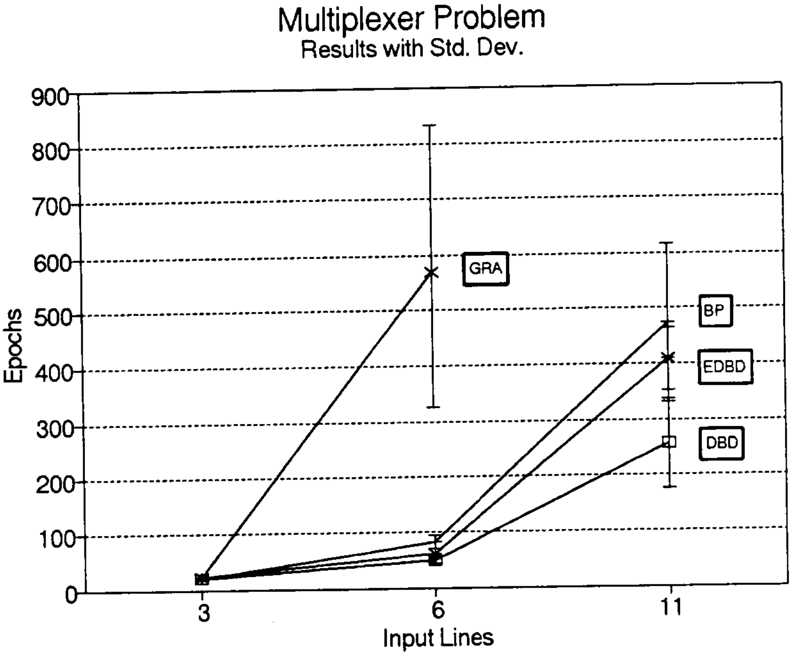


Figure 5.10: Graph of Best Average Convergence Times with Std. Dev. (from Sets of Runs with 100% Convergence) for the Multiplexer Problem.

all three problem sizes, the DBD algorithm produced better results than either the BP or the EDBD. For the 3-bit case, the advantage was small (19.6 vs 20.9 for BP), but for the 11 bit problem, the difference was larger (303.4 vs. 473.6 for BP). The EDBD algorithm produced results which were similar to, but not quite as good as, the DBD. The use of predefined weight values was of some benefit in reducing convergence times in the multiplexer trials, but only for the 11 bit problem size. For example, the EDBD results were reduced from an average best value of 408.4 to 329.8. The standard BP algorithm showed little change between the two methods, requiring 473.6 epochs with random initial weights and 473.4 with the grid-assigned weights. Figure 5.10 displays the best average convergence times (± one standard deviation) from sets of runs with 100% convergence for the multiplexer problems. The GRA produces significantly worse results, with

Algorithm	**Lrate**	**Momentum**				
	(η)	0.1	0.3	0.5	0.7	0.9
Back-Prop	0.1	4445	3483	2494	1515	649
	0.3	1391	1166	841	535	394
	0.5	898	703	512	354	690_1
	0.7	643	505	373	257	643_3
	0.9	502	395	296	180	329_5
	1.2	378	300	231	133	123_8
	1.5	304	243	171	112	109_8
	2.0	229	188	131	115	$—_{11}$
Sig: 1.1	0.7	562	455	311	220	306_5
GRA	0.5	308	246	189	131	89_7
	0.7	306	245	193	113	95_7
DBD	0.3	234	186	137	95	206_4
	0.5	251	184	136	94	82_6
	0.7	263	180	134	98	180_5
Sig: 1.1	0.5	237	167	123	85	75_7
EDBD	0.3	261	290	314	310_1	327_1
	0.5	167	229	191	246	230_2
	0.7	615_2	143_1	165_1	190	129_4
Sig: 1.1	0.5	149	189	173	192_2	184_7

Table 5.15: Convergence Times for the 4-Bit Encoder/Decoder Problem Using the BP, GRA, DBD, EDBD Algorithms.

a large variance for the 6-bit case. The other algorithms produce better results, with the DBD algorithm notably better on the 11-bit problem size.

In summary, the multiplexer is shown to be much easier for a network trained with BP to learn than the XOR/Parity problem. The GRA was again found to be ineffective. The use of a modified sigmoid coefficient was not found to effective in producing lower convergence times. The preset weights resulted in lower convergence times only in the 11 bit case. The EDBD achieved better convergence times than BP on the larger (6 and 11 bit) cases. DBD produced the best results for all three problem sizes.

Algorithm	Lrate	Momentum				
	(η)	0.1	0.3	0.5	0.7	0.9
Back-Prop	0.1	10811	8391	5589	3470	1432_1
	0.3	3397	2648	1935	1162	1717_8
	0.5	2045	1593	1168	611	$—_{11}$
	0.7	1450	1145	806	476_1	$—_{11}$
	0.9	1163	950	557	414_1	$—_{11}$
	1.2	946	619	405_2	$—_{11}$	$—_{11}$
	1.5	702	470	375	405_8	$—_{11}$
	2.0	467_1	388_2	378_7	$—_{11}$	$—_{11}$
Sig: 1.1	0.7	1178	929	623	467_1	$—_{11}$
GRA	0.5	758	586	382	260_1	$—_{11}$
	0.7	556	477	376_1	263_4	$—_{11}$
DBD	0.3	492	406	259	182	$—_{11}$
	0.5	526	416	289	232	$—_{11}$
	0.7	552	401	268	286	$—_{11}$
Sig: 1.1	0.5	412	313	247	177_1	$—_{11}$
EDBD	0.3	598	621	663	645_2	869_6
	0.5	408	457	486	496_6	$—_{11}$
	0.7	329	340	384_1	443_9	$-_{11}$
Sig: 1.1	0.5	342	417	405	390_6	$—_{11}$

Table 5.16: Convergence Times for the 8-Bit Encoder/Decoder Problem Using the BP, GRA, DBD, EDBD Algorithms.

5.3 Encoder/Decoder Problem Results

The encoder/decoder problem was run for the three cases of 4, 8, and 10 inputs. The error criterion used was modified from the total sum of squared error to a maximum error per pattern criterion, as described previously. However, one additional problem was noticed. The encoder problems utilize multiple output nodes. There is an error associated with each output upon presentation of each pattern. Using the original sum of squared error criterion, the error terms from each output node are summed together, and this value must be brought below the tolerance value in order for the network to be considered trained. Again, large number of output nodes could overwhelm any small error tolerance, even though the correct outputs were being generated. The error criterion was modified so that no output node could produce an error greater than the error tolerance

Algorithm	Lrate	Momentum				
	(η)	0.1	0.3	0.5	0.7	0.9
Back-Prop	0.1	5286	4069	2884	1655	1960_5
	0.3	1749	1404	910	550	$—_{11}$
	0.5	1078	759	531	428	$—_{11}$
	0.7	675	525	384	455	$—_{11}$
	0.9	523	392	349	320_6	$—_{11}$
Sig: 1.1	0.7	555	417	361	352_4	$—_{11}$
GRA	0.5	535	406	267	248_3	$—_{11}$
	0.7	482	393	270	257_6	$—_{11}$
DBD	0.3	274	223	150	108	$—_{11}$
	0.5	289	194	153	124_1	$—_{11}$
	0.7	253	193	149	223_1	$—_{11}$
Sig: 1.1	0.5	210	163	131_1	143_3	$—_{11}$
EDBD	0.3	353	384	418	472	$—_{11}$
	0.5	221	238	252	389_2	$—_{11}$
	0.7	160	174	230	363_5	$—_{11}$
Sig: 1.1	0.5	201	189	218	286_5	$—_{11}$

Table 5.17: Convergence Times for the 10-Bit Encoder/Decoder Problem Using the BP, GRA, DBD, EDBD Algorithms

specified on a per pattern basis. Thus, to correctly classify an input pattern, a network must produce a result on each of its output nodes which differs from the correct output by no more than the error tolerance specified.

The results for the 4-2-4 encoder problem are shown in Table 5-15. Many of the same characteristics observed in the XOR and multiplexer problems occur in the encoder/decoder problem, as well. These are: the approximately linearly decreasing convergence times with increasing momentum and learning rates, similar limits to the effectiveness of the increases, reductions in convergence times when using a modified sigmoid coefficient (using a value of 1.1), and improved convergence times with the use of the DBD algorithm. The EDBD had higher (worse) convergence times than BP with the 4-bit problem. Similar run characteristics were observed with the 8-bit problem (Table 5.16) and the 10-bit problem (Table 5.17), except that EDBD also had better convergence times than BP. The best convergence times for the algorithms for all encoder/decoder prob-

Size	Algorithm	#	Mean	Std Dev	Min	Max	η	α	Other Parameters
4-Bit	Std. Back-Prop.	11	112.3	29.1	86	158	1.5	0.7	
	Mod. Sig.		108.9	35.1	66	187	1.5	0.7	Sig. Coeff (D): 1.1
	Gradient Reuse		113.1	45.3	84	173	0.9	0.7	Upd. Coeff: 0.1, Max Lrate: 1.5
	Delta-Bar-Delta		59.1	27.5	35	127	0.7	0.5	κ: .265, ϕ: 0.1, θ: 0.3
	Mod. Sig.		55.5	20.1	33	78	0.7	0.5	κ: .265, ϕ: 0.1, θ: 0.3, D:1.2
	Extended DBD		139.9	64.5	64	265	4.0	0.1	κ_l: .5, ϕ_l: 0.1, γ_l: 0.1, θ: 0.3
	Mod. Sig.		89.4	66.5	32	235	2.0	0.1	κ_m: .05, ϕ_m: .2, γ_m: 0.3, D: 1.1
8-Bit	Std. Back-Prop.	11	371.4	53.7	306	448	1.5	0.5	
	Mod. Sig.		432.5	107.4	318	719	1.2	0.5	Sig. Coeff: 1.05
	Gradient Reuse		259.9	54.3	201	483	0.5	0.7	Upd. Coeff: 0.1, Max Lrate: 1.5
	Delta-Bar-Delta		103.1	24.7	75	160	0.5	0.7	κ: .3, ϕ: 0.4, θ: 0.1
	Mod. Sig.		95.6	16.9	72	119	0.5	0.7	κ: .3, ϕ: 0.4, θ: 0.1, D: 1.05
	Extended DBD		138.6	47.4	69	236	1.4	0.2	κ_l: .3, ϕ_l: 0.04, γ_l: 0.05, θ: 0.15
	Mod. Sig.		118.6	35.3	69	143	1.2	0.2	κ_m: .05, ϕ_m: .1, γ_m: 0.1, D: 1.1
10-Bit	Std. Back-Prop.	11	349.4	61.4	250	469	0.9	0.5	
	Mod. Sig.		353.3	68.4	244	523	0.9	0.5	Sig. Coeff: 1.05
	Gradient Reuse		266.5	38	214	328	0.5	0.5	Upd. Coeff: 0.1
	Delta-Bar-Delta		96.5	16.3	79	128	0.5	0.5	κ: .3, ϕ: 0.5, θ: 0.1
	Mod. Sig.		96.4	17.1	72	131	0.5	0.5	κ: .3, ϕ: 0.5, θ: 0.1, D: 1.05
	Extended DBD		120.2	36.2	74	174	1.4	0.2	κ_l: .3, ϕ_l: 0.04, γ_l: 0.05, θ: 0.15
	Mod. Sig.		129.5	37	71	198	1.4	0.2	κ_m: .05, ϕ_m: .1, γ_m: 0.1, D: 1.05

Table 5.18: Summary of Best Average Convergence Times (in Epochs) from Sets of Runs with 100% Convergence, for the Encoder/Decoder Problem.

lem sizes can be found in Table 5.18. As shown in this table, the DBD algorithm achieved the lowest convergence times for all three problem sizes. The GRA achieved similar results to BP (113.1 vs 112.3 for the 4-bit case) or better than BP (259.9 vs 371.4 for the 8-bit case, and 266.5 vs 349.4 for the 10 bit case). The EDBD generally perfomed slightly worse than the DBD, and for the 4-bit input problem it had the worst performance results. For the 8 and 10 input cases, it ranked second to the DBD. These results are also shown in Figure 5.11, which displays the best average convergence times (± one standard deviation) from sets of runs with 100% convergence for the encoder/decoder problem.

It was found that small increases to the value of the sigmoid sharpness coefficient improved performance, but usually only slightly. Values larger than 1.05 or 1.1 tended to result in large numbers of non-converging runs. After additional testing, it seemed apparent that the use of an elevated sharpness coef-

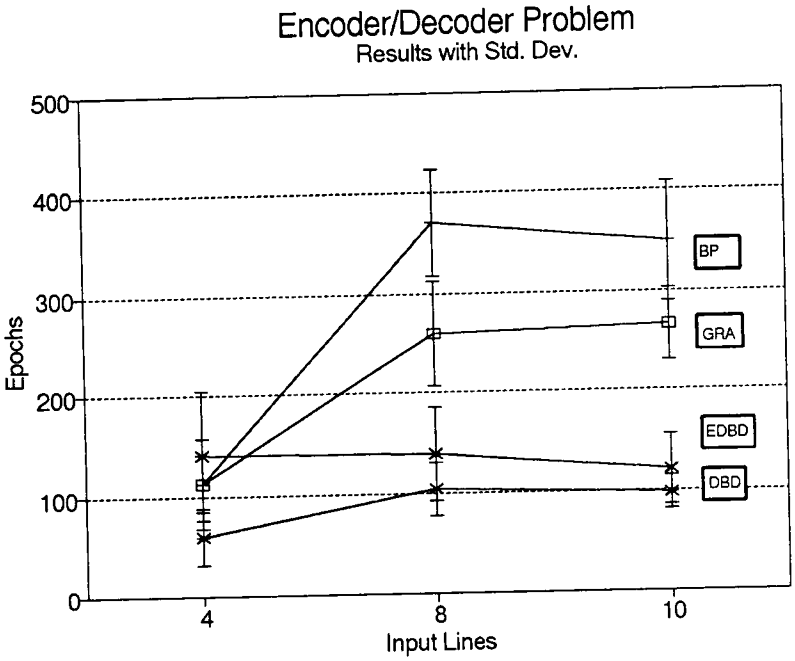


Figure 5.11: Graph of Best Average Convergence Times with Std. Dev. (from Sets of Runs with 100% Convergence) for the Encoder/Decoder Problem.

ficient with the BP algorithm does little more than boost the learning rate (and hence the weight adjustment) by its value. Thus, similar effects can be achieved using slightly greater learning rates, instead. On several trials the learning rate, sigmoid coefficient combination was set to (1.0, 1.1) and compared to runs made with settings of (1.1, 1.0). Although slight differences in convergence times (1 or 2 epochs out of 100) were seen, the results were essentially identical. It appears that the use of the sharpened sigmoid can in most cases be replaced with a correspondingly increased learning rate.

In summary, the modification of the error criterion to a maximimum permissible error per pattern at each output node allowed a single error criterion to be used on all problems sizes. Added processing nodes in the hidden layer of the 10-5-10 encoder problem allowed the network to converge to a solution in

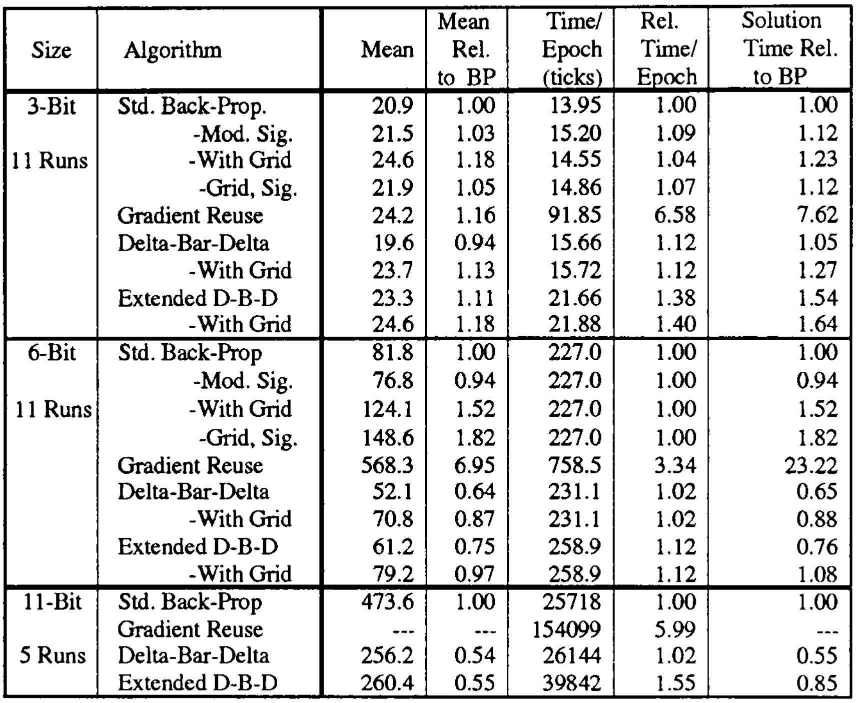
Size	Algorithm	Mean	Mean Rel. to BP	Time/ Epoch (ticks)	Rel. Time/ Epoch	Solution Time Rel. to BP
2-Bit	Std. Back-Prop.	181.8	1.00	4.71	1.00	1.00
	-High Lrate	76.5	0.42	4.71	1.00	0.42
11 runs	-Mod. Sig.	73.0	0.40	4.76	1.01	0.41
	-With Grid	40.0	0.22	4.71	1.00	0.22
	-Grid, Sig.	36.6	0.20	4.74	1.01	0.20
	Gradient Reuse	165.0	0.91	31.38	6.66	6.05
	Delta-Bar-Delta	83.2	0.46	5.66	1.20	0.55
	-With Grid	74.6	0.41	5.66	1.20	0.49
	Extended D-B-D	43.5	0.24	8.73	1.54	0.37
	-With Grid	31.7	0.17	8.73	1.54	0.27
3-Bit	Std. Back-Prop	206.7	1.00	14.21	1.00	1.00
	-Mod. Sig.	133.6	0.65	14.83	1.04	0.65
11 Runs	-With Grid	104.2	0.50	14.23	1.00	0.50
	-Grid, Sig.	95.2	0.46	14.36	1.01	0.47
	Gradient Reuse	1387.3	6.71	85.37	6.01	40.32
	Delta-Bar-Delta	172.8	0.84	15.75	1.11	0.93
	-With Grid	141.3	0.68	15.65	1.10	0.75
	Extended D-B-D	77.6	0.38	22.03	1.41	0.41
	-With Grid	58.7	0.28	22.16	1.42	0.40
4-Bit	Std. Back-Prop	3044.6	1.00	36.73	1.00	1.00
	Gradient Reuse	3894.0	1.28	203.66	5.54	1.28
5 Runs	Delta-Bar-Delta	638.4	0.21	38.22	1.04	0.22
	Extended D-B-D	539.2	0.18	50.47	1.37	0.24

Figure 5.12: Algorithm Timing Comparison for the XOR/Parity Problem.

fewer epochs. This result implies that improved network convergence times (for other problem types, as well) could result from the incorporation of additional hidden layer nodes.

5.4 Timing Results

While not critical to the parallel implementation of neural networks, the simulation of ever-larger problems and networks on sequential machines will require that some attention be given to the increasing time requirements. In an effort to quantify computational requirements, timing runs were made for each of the



Size	Algorithm	Mean	Mean Rel. to BP	Time/ Epoch (ticks)	Rel. Time/ Epoch	Solution Time Rel. to BP
3-Bit	Std. Back-Prop.	20.9	1.00	13.95	1.00	1.00
	-Mod. Sig.	21.5	1.03	15.20	1.09	1.12
11 Runs	-With Grid	24.6	1.18	14.55	1.04	1.23
	-Grid, Sig.	21.9	1.05	14.86	1.07	1.12
	Gradient Reuse	24.2	1.16	91.85	6.58	7.62
	Delta-Bar-Delta	19.6	0.94	15.66	1.12	1.05
	-With Grid	23.7	1.13	15.72	1.12	1.27
	Extended D-B-D	23.3	1.11	21.66	1.38	1.54
	-With Grid	24.6	1.18	21.88	1.40	1.64
6-Bit	Std. Back-Prop	81.8	1.00	227.0	1.00	1.00
	-Mod. Sig.	76.8	0.94	227.0	1.00	0.94
11 Runs	-With Grid	124.1	1.52	227.0	1.00	1.52
	-Grid, Sig.	148.6	1.82	227.0	1.00	1.82
	Gradient Reuse	568.3	6.95	758.5	3.34	23.22
	Delta-Bar-Delta	52.1	0.64	231.1	1.02	0.65
	-With Grid	70.8	0.87	231.1	1.02	0.88
	Extended D-B-D	61.2	0.75	258.9	1.12	0.76
	-With Grid	79.2	0.97	258.9	1.12	1.08
11-Bit	Std. Back-Prop	473.6	1.00	25718	1.00	1.00
	Gradient Reuse	---	---	154099	5.99	---
5 Runs	Delta-Bar-Delta	256.2	0.54	26144	1.02	0.55
	Extended D-B-D	260.4	0.55	39842	1.55	0.85

Figure 5.13: Algorithm Timing Comparison for the Multiplexer Problem.

algorithms for each of the problem sizes. The relative time required per epoch of processing is shown in Figure 5.12 for the XOR problem, in Figure 5.13 for the Multiplexer problem, and in Figure 5.14 for the encoder/decoder problem. The relative time required per epoch is a measure of the computation time required for a complete pass through the training set, including all weight updates. It is measured relative to BP for the same size problem, and thus reflects any additional calculation required for the algorithms. As shown in Figure 5.12, for the 2-bit XOR, DBD requires 20% more time per epoch in order to implement the adaptive calculations. The EDBD requires 54% more processing time. Both of these algorithms converged to a solution in much less time than did the low-

Size	Algorithm	Mean	Mean Rel. to BP	Time/ Epoch (ticks)	Rel. Time/ Epoch	Solution Time Rel. to BP
	Std. Back-Prop.	112.3	1.00	11.65	1.00	1.00
4-Bit	-Mod. Sig.	108.9	0.97	11.69	1.00	0.97
	Gradient Reuse	113.1	1.01	100.69	8.64	8.70
	Delta-Bar-Delta	59.1	0.53	13.73	1.18	0.62
	-Mod. Sig.	55.5	0.49	13.84	1.19	0.59
	Extended D-B-D	139.9	1.25	19.55	1.68	2.09
	-Mod. Sig.	89.4	0.80	19.87	1.71	1.36
	Std. Back-Prop	371.4	1.00	50.87	1.00	1.00
8-Bit	-Mod. Sig.	432.5	1.16	51.03	1.00	1.17
	Gradient Reuse	259.9	0.70	362.03	7.12	4.98
	Delta-Bar-Delta	103.1	0.28	54.60	1.07	0.30
	-Mod. Sig.	95.6	0.26	54.78	1.08	0.28
	Extended D-B-D	138.6	0.37	69.32	1.36	0.51
	-Mod. Sig.	118.6	0.32	69.48	1.37	0.44
	Std. Back-Prop	349.4	1.00	102.61	1.00	1.00
10-Bit	-Mod. Sig.	353.3	1.01	103.10	1.00	1.02
	Gradient Reuse	266.5	0.76	519.61	5.06	3.86
	Delta-Bar-Delta	96.5	0.28	111.28	1.08	0.30
	-Mod. Sig.	96.4	0.28	111.96	1.09	0.30
	Extended D-B-D	120.2	0.34	140.29	1.37	0.47
	-Mod. Sig.	129.5	0.37	140.91	1.37	0.51

Figure 5.14: Algorithm Timing Comparison for the Encoder/Decoder Problem.

learning rate BP algorithm. The solution time relative to BP is shown in the last column. This value is the product of the number of epochs required to converge (relative to BP) and the computation time per epoch (relative to BP). This column shows, for example, for the 2-bit XOR, that DBD requires 55% of the time to reach convergence that BP does. The relative times vary from problem to problem and differ by size, but with the exception of the 3-bit multiplexer problem, the DBD algorithm consistently requires less time to converge than does BP on the same problem size. The timing results may be of little relevance to parallel implementations, but may serve as a basis for estimating execution times on sequential machines for scaled up problem sizes. They also serve as

# of Input Lines	Expected Growth		Actual Growth	
	$2^n\,(n+1)^2$	Normalized		Normalized
2	36	1.0	4.71	1.0
3	128	3.6	14.21	3.0
4	400	11.1	36.73	7.7
6	3136	87.1	226.95	47.7
11	147456	4096.0	25718.00	5402.9

Table 5.19: Scaling: Relationship of Network Size to Relative Solution Times for n-n-1 Networks with 2^n Inputs.

an additional performance measure for the algorithms tested. For the XOR and multiplexer problems, an exponential increase in time is required for networks to learn larger problems is shown in Table 5.19. An estimated rate of increase, based upon 2^n inputs and $(n+1)^2$ weights is also included for comparison. The results are not exact, but the relative rate of increase is similar. Per-epoch processing time increases in the encoder/decoder problem at a polynomial rate. The n input patterns are applied each epoch, and there are $2n\,log\,n$ weight updates per pattern, resulting in an overall $O(n^2)$ increase in processing time as the number of input patterns is increased. A graph of the increased computation requirements needed as the network size is increased is shown in Figure 5.16. The per-epoch computation requirements for XOR and the multiplexer problem are nearly equal at $n = 3$, and increase at similar rates. The difficulty of the XOR problem with respect to the multiplexer may be seen in Figure 5.15. Even as the per-epoch computational requirements of the problems are rising with increasing size, the number of epochs required to converge is increasing as well. This is particularly

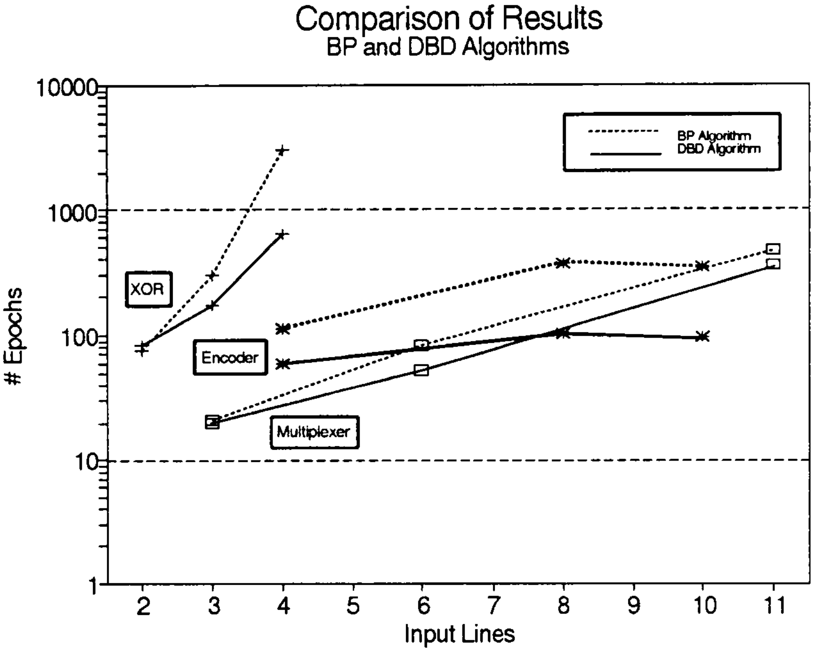


Figure 5.15: Comparison of Convergence Times for BP and DBD.

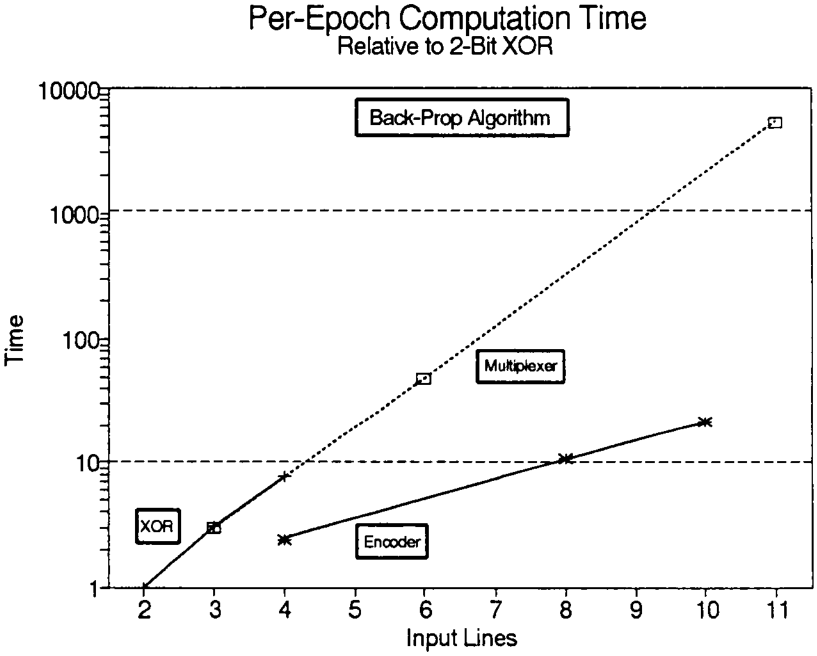


Figure 5.16: Relative Per-Epoch Computation Times Required for BP.

verge is noted on the XOR problem. This increase in processing time demonstrates the need for parallel implementation of neural network architectures. The exponential increase is most noticable for the 11 bit multiplexer problem, in which each epoch requires approximately 5400 times the amount of processing as for the 2-bit XOR.

Chapter 6

Conclusions

In this study, neural networks using the back-propagation algorithm for weight adjustment during training have been shown to be capable of learning solutions for three different types of benchmark problems: the XOR/parity, multiplexer, and the encoder/decoder. Each problem was scaled up to three different sizes. The BP algorithm and variants based upon heuristics were applied to each of these cases.

It was confirmed that the BP algorithm can exhibit slow convergence rates, and converges to non-optimal local minima. It has also been shown how increases to both learning rate and momentum terms can reduce convergence times without seriously increasing the number of non-converging runs. Limits to the effectiveness of these increases are also shown. For BP, convergence times are shown to decrease approximately linearly with linear increases in either momentum or learning rate. Momentum levels of 0.9 and higher produced increasing levels of non-converging runs, particularly when coupled with high learning rate values. The learning rate parameter may be set to large values (occassionally approaching 10.0) before non-converging runs becomes a significant problem. This characteristic of BP was observed in the three problems tested, and for each of their sizes.

The Gradient Reuse Algorithm was not found to perform as well as the standard BP algorithm for the XOR and Multiplexer problems. It achieved similar convergence times for the smaller problems, but did not appear to scale well. It was susceptible to non-convergence caused by even small changes in input parameter settings. It did produce better results in the encoder/decoder problem than BP, but this was achieved at much greater computational cost (Figure 5.11). The failure of the GRA would indicate that error gradients calculated at one point on the error surface are not generally useful in determining favorable weight adjustments at a different point on the error surface.

The use of an increased sigmoid coefficient resulted in slightly improved convergence times, although the best results obtained were usually very similar to the best results achieved with the BP algorithm with a corresponding increase in the learning rate coefficient. Tests on the encoder/decoder problem in which η was increased by a factor of D, and the sigmoid coefficient reset to 1.0, resulted in essentially the same convergence times. Thus, the use of increased sigmoid coefficient settings do not provide any significant improvement in convergence times as compared to the standard BP algorithm.

The use of pre-defined weights to pre-partition input parameter space in order to speed up convergence worked well with the 2-Bit XOR problem (producing reductions of 50% in convergence times), and with the 11-Bit multiplexer problem using the adaptive algorithms (20% reductions) . It seemed well suited for the XOR problem, dividing the n-dimensional hypercube into 2^n classification regions which could then be utilized by the network's subsequent layers, resulting in improved classification results in fewer epochs. However, it did not perform well on the 3 and 4 bit XOR problems. The use of bipolar values for network inputs and nodal outputs, coupled with non-zero initial weight settings might avoid the inherent inability of the BP algorithm to adjust weights (and hence, learn) whenever zero inputs are present. The reduced number of randomly assigned weight values (in the hidden layer) may also cause this modification to be more sensitive to initial conditions (weight values) than BP.

The scale-up from the small initial problem sizes resulted in the recognition of a problem associated with the use of the total sum of squared error criterion to determine when a network has successfully been trained. The cause of the problem is due to the error summation over all input patterns and across all output nodes. As the number of input patterns increases, this sum increases as well, so that a particular error specification which is useful for a 2-bit problem becomes unattainable (in reasonable processing time) with even a 6 or 11-bit problem. The solution was to change the error criterion to a 'maximum permissable error' per pattern, measured at each output node. The error setting, then, does not require modification as a network size is increased in order to handle a larger number of input patterns. Similarly, it requires no modification as the number of output nodes increases, which might occur with an increase in the number of output classes, such as with the encoder/decoder problem.

The adaptive algorithms, particularly DBD, are capable of reducing convergence times as compared to BP. Both the DBD and the EDBD can take advantage of very large peak-learning rates (up to 30.0 and higher) to increase appropriate weight changes and reduce subsequent convergence times with only small increases in added computational times. These large learning rate values do not increase non-convergence rates significantly because the adaptive mechanism allows for very rapid reductions to take place whenever the shape of the error surface requires it. Thus, the adaptive step size coefficients for each weight in the network can be individually adjusted to allow the network to swiftly move across the relatively flat portions of the error curve, but then self-adjust in order to closely track small changes when this is required, as well. Although a great deal of parameter tuning on the 2-bit XOR problem resulted in $\sim$ 50% reduction in convergence times, the overall similarity of results with DBD and EDBD indicates that the importance of a trainable momentum parameter is not as critical to the performance of the algorithm as is an adaptable learning rate parameter. High rates of momentum (0.9, or 0.95) with the DBD algorithm produced the best results.

The timing runs provide an indication of the rapid increase in computation time with even small increases in problem size. Not only do convergence times increase with increased problem size, the per-epoch computation time increases as well. It seems that parallel network implementations will be required in order to minimize this increase in computational requirements.

The results of this study indicate that the weight update equation for the BP algorithm (and for the additional heuristics) needs to be modified to use a normalized weight update value rather than a sum of the per-pattern weight updates.

The error criterion for determining successful learning also needs to be modified from a 'total sum of squared error' summed over all input patterns over all output nodes to a maximum permissible per-pattern error criterion at each output node. The implementation of this modification and the normalized weight update allows larger networks to be defined which can be trained with a larger number of input patterns (and with a larger number of output nodes) without requiring a change in the error criterion.

Use of the adaptive algorithms resulted in reduced convergence times across all problem types and sizes, with the differences most noted with larger problem sizes. The similar performance of the DBD and the EDBD favors the implementation of the DBD algorithm. It is easier to implement and requires less processing time per epoch than the EDBD. Its chief advantage is that it requires the specification of only 3 additional input parameters (4 with a maximum learning rate limit) compared to 9 with the EDBD. It has been shown that for a small increase in computation time, the DBD adaptive algorithm can achieve significant reductions in convergence times, with little or no increase in non-convergence rates. Although improved settings of the learning rate and momentum coefficients can also reduce convergence times on the BP algorithm, the introduction of an adaptive algorithm with a small set of input parameters can be used to significantly improve upon even the best results obtainable with the standard back-propagation algorithm.

BIBLIOGRAPHY

Bibliography

[Ackley 85] Ackley, D.H., Hinton, G.E., Sejnowski, T.J., A Learning Algorithm for Boltzmann Machines, *Cognitive Science,* **9**, 147-169, 1985.

[Anderson 88] Anderson, J.A., Rosenfeld, E. [Eds.], *Neurocomputing: Foundations of Research,* MIT Press, Cambridge MA, 1988.

[Barto 83] Barto, A.G., Sutton, R.S., Anderson, C.W., Neuron-like adaptive elements that can solve difficult learning control problems, *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC-13**, 834-846, 1983.

[Battiti 90] Battiti, R., Optimization Methods for Back-Propagation: Automatic Parameter Tuning and Faster Convergence, *Proceedings of the International Joint Conference on Neural Networks,* **I**, 593-596, 1990.

[Baum 88a] Baum, E.B., Moody, J., Wilezek. F., Internal Representations for Associative Memory, *Biological Cybernetics,* **59**, 217-228, 1988.

[Baum 88b] Baum, E.B., On the Capabilities of Multilayer Perceptrons, *Journal of Complexity,* **4**, 193-215, 1988.

[Block 62] Block, H.D., The Perceptron: a model for brain functioning, *Reviews of Modern Physics,* **34**, 123:135, 1962.

[Carpenter 87] Carpenter, G.A., Grossberg, S., ART-2: Self-organization of stable category recognition codes for analog input patterns, *Applied Optics,* **26**, 4919-4930, 1987.

[Cater 87] Cater, J.P., Successfully using Peak Learning Rates of 10 (and greater) in Back-Propagation Networks with the Heuristic Learning Algorithm, *Proceedings of the First International Conference on Neural Networks,* **II**, 645-651, 1987.

[Chen 90] Chen, J.R., and Mars, P., Stepsize Variation Methods for Accelerating the Back-Propagation Algorithm, *Proceedings of the International Joint Conference on Neural Networks,* **I**, 601-604, 1990.

[Dahl 87] Dahl, E.D., Accelerated Learning Using the Generalized Delta Rule, *Proceedings of the 1st International Conference on Neural Networks,* **II**, 523-530, 1987.

[Duda 73] Duda, R.O., and Hart, P.E., *Pattern Classification and Scene Analysis,* Wiley, New York, 1973.

[Fahlman 88] Fahlman, S.E., Faster-Learning Variations on Back-Propagation: An Empirical Study, *Proceedings of the 1988*

Summer School on Connectionist Models, Carnegie-Mellon Univ., 38-51, 1988.

[Fukushima 83] Fukushima, K., Miyake, S., Ito, T., Neocognitron: a neural network model for a mechanism of visual pattern recognition, *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC-13**, 826-834, 1983.

[Geman 84] Geman, S., and Geman, D., Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **PAMI-6**, 721-741, 1984.

[Grossberg 88] Grossberg, S. [Ed], *Neural Networks and Natural Intelligence,* MIT Press, Cambridge MA, 1988.

[Hebb 49] Hebb, D.O., *The Organization of Behavior,* Wiley, New York, 1949.

[Hecht-Nielsen 87] Hecht-Nielsen, R., Kolmogorov's Mapping Neural Network Existence Theorem, *Proceedings of the First International Conference on Neural Networks,* **III**, 11-13, 1987.

[Hecht-Nielsen 89] Hecht-Nielsen, R., Theory of the Backpropagation Neural Network, *Proceedings of the International Joint Conference on Neural Networks,* **I**, 593-605, 1989.

[Hecht-Nielsen 90] Hecht-Nielsen, R., *Neurocomputing,* Addison Wesley, New York, 1990.

[Higashino 90] Higashino, J., deGreef, B.L., Persoon, E.H.J., Numerical Analysis and Adaption Method for Learning Rate of Back Propagation, *Proceedings of the International Joint Conference on Neural Networks,* **I**, 627-630, 1990.

[Hinton 86] Hinton, G.E., and Sejnowski, T.S., Learning and Relearning in Boltzmann Machines in *Parallel Distributed Processing,* **I**, 282-317, MIT Press, Cambridge MA, 1986.

[Hinton 87] Hinton, G.E., *Connectionist Learning Procedures,* Technical Report CMU-CS-87-115, Carnegie-Mellon University, Computer Science Dept., Pittsburgh PA, 1987.

[Hopfield 82] Hopfield, J.J., Neural Networks and Physical Systems with Emergent Collective Computational Abilities, *Proceedings of the National Academy of Science,* **79**, 2554-2558, 1982.

[Hopfield 85] Hopfield, J.J., and Tank, D.W., Neural Computation of Decisions in Optimization Problems, *Biological Cybernetics,* **52**, 141-152, 1985.

[Hopfield 86] Hopfield, J.J., and Tank, D.W., Neural Computing with Neural Circuits: A Model, *Science,* **233**, 625-633, 1986.

[Huang 87] Huang, W.Y., and Lippmann, R., Comparisons Between Neural Net and Conventional Classifiers, *Proceedings of the 1st International Conference on Neural Networks,* **IV**, 485-493, 1987.

[Hush 88] Hush, J.R., and Salas, J.M., Improving the Learning Rate of Back-Propagation with the Gradient Reuse Algorithm, *Proceedings of the International Conference on Neural Networks,* **I**, 639-642, 1988.

[Izui 90] Izui, Y., and Pentland, A., Speeding Up Back Propagation, *Proceedings of the International Joint Conference on Neural Networks,* **I**, 639-642, 1990.

[Jacobs 88] Jacobs, R.A., Increased Rates of Convergence Through Learning Rate Adaptation, *Neural Networks,* **1**, 295-307, 1988.

[Kirkpatrick 83] Kirkpatrick, S., Gelat, C.D. Jr., and Vecchi, M.P., Optimization by Simulated Annealing, *Science,* **220**, 671-680, 1983.

[Kohonen 82] Kohonen, T., Self-Organized Formation of Topologically Correct Feature Maps, *Biological Cybernetics,* **43**, 59-69, 1982.

[Kohonen 84] Kohonen, T., *Self-Organization and Associative Memory,* Springer-Verlag, Berlin, 1984.

[Kohonen 88] Kohonen, T., An Introduction to Neural Computing, *Neural Networks,* **1**, 3-16, 1988.

[Kohonen 89] Kohonen, T., Speech Recognition Based on Topology-Preserving Neural Maps, in I. Aleksander [Ed.] *Neural Computing Architectures,* Bradford Books/MIT Press, Cambridge MA, 1989.

[Kolen 90] Kolen, J.F., Pollack, J.B., Back Propagation is Sensitive to Initial Conditions, *Technical Report TR 90-JK-BPSIC,* Ohio State University, 1990.

[Kosko 88] Kosko, B., Bidirectional Associative Memories, *IEEE Transactions on Systems, Man, and Cybernetics,* **18**, 49-60, 1988.

[Lippmann 87a] Lippmann, R.P., An Introduction to Computing with Neural Networks, *IEEE ASSP Magazine,* **4**, 4-22, 1987.

[Lippmann 87b] Lippmann, R.P., and Gold, B., Neural-Net Classifiers Useful for Speech Recognition, *Proceedings of the 1st International Conference on Neural Networks,* **IV**, 485-493, 1987.

[McClelland 86] McClelland, J.L., and Rumelhart, D.E. [Eds.], *Explorations in Parallel Distributed Processing: Models, Programs and Exercises,* MIT Press, Cambridge MA, 1986.

[McCulloch 43] McCulloch, W.S., and Pitts, W., A Logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics,* **5**, 115-133, 1943.

[Minai 90] Minai, A.A., and Williams, R.D., Acceleration of Back Propagation through Learning Rate and Momentum Adaptation, *Proceedings of the International Joint Conference on Neural Networks,* **I**, 676-679, 1990.

[Minsky 69] Minsky, M., and Papert, S., *Perceptrons,* MIT Press, Cambridge MA, 1969.

[Nilson 65] Nilson, N.J., *Learning Machines: Foundatations of Trainable Pattern Classifying Systems,* McGraw-Hill, New York, 1965.

[Oblow 90] Oblow, E., *personal communication,* 1990.

[Pao 89] Pao, Y.H., *Adaptive Pattern Recognition and Neural Networks,* Addison-Wesley, New York, 1989.

[Parker 87] Parker, D.B., Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second Order Direct Propagation, and Second Order Hebbian Learning, *Proceedings of the First International Conference on Neural Networks,* **I**, 593-600, 1987.

[Rosenblatt 62] Rosenblatt, F., *Principles of Neurodynamics,* Spartan Books, Washington DC, 1962.

[Rumelhart 86a] Rumelhart, D.E., Hinton, G.E., and Williams, R.J., Learning Internal Representations by Error Propagation, in Rumelhart & McClelland [Eds.] *Parallel Distribued Processing: Explorations in the Microstructure of Cognition,* **I**, pp. 318-362, MIT Press, Camridge MA, 1986.

[Rumelhart 86b] Rumelhart, D.E., and McClelland, J.L., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* **I & II**, MIT Press, Cambridge MA, 1986.

[Sklansky 81] Sklansky, J. and Wassel, G.N., *Pattern Classifiers and Trainable Machines,* Springer-Verlag, New York, 1981.

[Sejnowski 86] Sejnowski, T.J., Rosenberg, C.R., NETtalk: a parallel network that learns to read aloud, *John Hopkins Univ. EE & CS Tech. Report JHU/EECS-86/01,* Jan 1986.

[Stornetta 87] Stornetta, W.S., and Huberman, B.A., An Improved Three-Layer Back-Propagation Algorithm, *Proceedings of the First International Conference on Neural Networks,* **II**, 637-643, 1987.

[Szu 87] Szu, H., and Hartley, R., Fast Simulated Annealing, *Physics Letters,* **122(3,4)**, 157-162, 1987.

[Tesauro 87] Tesauro, G., Scaling relationships in back-propagation learning: Dependence on training set size, *Complex Systems,* **2**, 367-372, 1987.

[Valiant 84] Valiant, L.G., A Theory of the Learnable, *Communications of the ACM,* **27**, 1134-1142, 1984.

[Vogl 88] Vogl, T.P., Mangis, J.K., Rigler, A.K., Zink, W.T., Akon, D.L., Accelerating the Convergence of the Back-Propagation Method, *Biological Cybernetics,* **59**, 257-263, 1988.

[Wasserman 89] Wasserman, P.D., *Neural Computing: Theory and Practice,* Van Nostrand Rheinhold, New York, 1989.

[Werbos 88] Werbos, P.J., Generalization of Backpropagation with Application to a Recurrent Gas Market Model, *Neural Networks,* **1**, 339-356, 1988.

[Widrow 60] Widrow, B., and Hoff, M.E., *Adaptive Switching Circuits,* IRE-WESCON Convention Record, 96-104, 1960.

[Widrow 85] Widrow, B., and Stearns, S.D., *Adaptive Signal Processing,* Prentice-Hall, Englewood Cliffs NJ, 1985.

[Widrow 87] Widrow, B., Winter, R., and Baxter, R., Learning Phenonema in Layered Neural Networks, *Proceedings of the International Conference on Neural Networks,* **II**, 411-429, 1987.

[Widrow 88] Widrow, B., and Winter, R., Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition, *IEEE Computer,* **21**, 25-39, 1988.

APPENDIX

Appendix A

Derivative of Sigmoid Function

$$out = f(net) = \frac{1}{1+\exp^{-net}} \quad where: \quad net = \sum_i w_i x_i + \theta$$

$$\begin{aligned}
\frac{\partial out}{\partial net} = f'(net) &= \exp^{-net}(1+\exp^{-net})^{-2} & \quad & Using: \\
&= \exp^{-net}\, out^2 & & out = (1+\exp^{-net})^{-1} \\
&= (\frac{1}{out} - 1)out^2 & & \exp^{-net} = \frac{1}{out} - 1 \\
&= out - out^2 & & \\
&= out(1 - out) & &
\end{aligned}$$

Vita

Kenneth S. Noggle was born 03 May 1956, in Oak Ridge, Tennessee. He graduated from Oak Ridge High School in 1974, and received his Bachelor of Arts in Computer Science from the University of Tennessee in 1980. From 1980 until 1988 he developed software systems for plant process control, real-time data acquisition, spectral analysis, and machine monitoring and control. In August 1988 he returned to the University of Tennessee to begin work on a Master's Degree. His interests include robotics, machine vision, adaptive pattern recognition, and applied artificial intelligence.