Performance and Hardware Compatibility of Backpropagation and Cascade Correlation Learning Algorithms

by

Brion K. Dolenko

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

Department of
Electrical and Computer Engineering
University of Manitoba

Winnipeg, Canada 1992

ISBN 0-315-81689-9

PERFORMANCE AND HARDWARE COMPATIBILITY OF BACKPROPAGATION AND CASCADE CORRELATION LEARNING ALGORITHMS

BY

BRION K. DOLENKO

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

An empirical analysis of performance of the backpropagation and cascade correlation neural network learning algorithms, and various minimization algorithms for use with each, is presented. Real-world pattern classification problems, including handwritten character recognition, are used in this analysis. A discussion of the hardware compatibility of the various algorithms is also presented, including a performance analysis of an analog hardware implementation of backpropagation. On one classification problem, namely cereal grain classification, neural network algorithms were found to result in better classification accuracy than a more traditional Gaussian maximum likelihood classifier, because of the neural networks' lack of assumptions about the problem. On the handwritten character recognition problem, and others as well, a complex, mathematically sound minimization algorithm was outperformed by a simple, less sound one, indicating the need for experimentation with several algorithms on a particular problem. In general, only small (in terms of number of processors) cascade correlation networks were found to perform better than similarly sized backpropagation-trained networks in terms of classification accuracy, due to the ease of learning in small cascade correlation networks, and the broader exploration of possible solutions done by large backpropagation-trained networks. In the analysis of the analog hardware implementation of backpropagation, it was found that the algorithm can overcome most of the shortcomings of the hardware.

Acknowledgements

The author wishes to thank Dr. Howard Card for his suggestions and support. Thanks are also given to the "regulars" in the VLSI lab, for their help in understanding some of the intricacies of the department's computing environment.

Drs. Hinton, Van Camp, and Steeg at the University of Toronto's Department of Computer Science developed and released the excellent neural network simulator that served as a foundation for this work.

Dr. Ahmadi at the University of Windsor's Department of Electrical Engineering made available the dataset of handwritten numerals, and Mike Neuman, working for the Food Science Department of the University of Manitoba, supplied the cereal grain database and some of his results.

Characteristics and variation estimates of the circuit components studied in this work were supplied by Dr. Christian Schneider.

This work was supported through funding from the Micronet Center of Excellence and the National Sciences and Engineering Research Council of Canada.

Contents

1 Introduction **1**
- 1.1 Purpose . . . 5
- 1.2 Problem . . . 5
- 1.3 Scope . . . 6

2 Background **8**
- 2.1 Typical Network Architecture and Operation . . . 8
- 2.2 The Backpropagation Learning Algorithm . . . 12
- 2.3 When to update the weights . . . 15
- 2.4 Problems with backpropagation, and fixes . . . 16
- 2.5 Backpropagation Enhancements . . . 18
- 2.6 The Cascade Correlation Algorithm . . . 24
- 2.7 Generalization . . . 28

3 Performance Analysis **30**
- 3.1 Methods for Performance Analysis . . . 30
  - 3.1.1 Methods for Type 1 Problems . . . 31
  - 3.1.2 Methods for Type 2 Problems . . . 33
- 3.2 Simulation Environment . . . 34
- 3.3 Results . . . 44
  - 3.3.1 Cereal Grain Classification . . . 44
  - 3.3.2 Handwritten Character Recognition . . . 53
  - 3.3.3 Parity and Sonar Signal Classification . . . 63

4 Hardware Considerations **72**
- 4.1 Computational and Storage Requirements . . . 72
- 4.2 Analysis of an Analog Hardware Backpropagation Circuit . . . 77
  - 4.2.1 Circuit Description . . . 79
  - 4.2.2 Circuit Modelling . . . 82
  - 4.2.3 Simulation Environment . . . 85
  - 4.2.4 Simulations Performed . . . 86
  - 4.2.5 Results from Learning in Analog Hardware . . . 87

5 Conclusions and Future Work 93

A Advanced Algorithms 96

B Xerion Parameter Defaults 100

List of Figures

2.1 Multi-layered Perceptron with Decision Regions 9
2.2 Typical Neuron Function 10
2.3 The Logistic Function 10
2.4 The Problem with Large Learning Rates 16
2.5 Logistic / Tanh Function Comparison 17
2.6 Steepest Descent / Conjugate Gradient Learning Comparison 21
2.7 The Cascade Correlation Algorithm 26
2.8 A Typical Network Constructed using Cascade Correlation 27

3.1 Typical Display of Xerion Simulator 35
3.2 Xerion Methodology for Network Training 36
3.3 Cross-Validation Method 39
3.4 MLP Generalization Performance on Grains Problem 48
3.5 MLP Training Time for Grains Problem 48
3.6 Cascor Generalization Performance on Grains Problem 50
3.7 Cascor Training Time for Grains Problem (tanh units) 51
3.8 MLP with Shortcuts Generalization Performance on Grains Problem 52
3.9 Sample of Handwritten Digit Data 54
3.10 Network Architecture for Handwritten Character Recognition 55
3.11 Learning Curves for AT&T Net 59
3.12 Learned Features for Handwritten Character Recognition 61
3.13 Thirty Worst Classified Test Digits 61

4.1 Schematic of Analog Circuit for Backpropagation Learning 80
4.2 Measured Gilbert Multiplier Characteristics 81
4.3 Gilbert Multiplier Model: $X1 * X2 => \tanh X1 * \tanh X2$ 83
4.4 Performance of 4-bit Parity Network with Multiplier Gain Variation . 88
4.5 Performance of Grain Classifier with Multiplier Gain Variation 88
4.6 Average Classification Accuracies with Limited Voltage Ranges 90

List of Tables

3.1 Output classification for single output neuron 41
3.2 Cross Validation Parameters for Cascade Correlation Grain Classifier 49
3.3 Summary of Best Average Grain Classification Generalization Results 53
3.4 AT&T Net: Simulation Parameters 58
3.5 Cross Validation Parameters for Cascade Correlation OCR Net 62
3.6 Average Learning Speeds: 4-bit parity network (8 hidden units) . . . 65
3.7 Original results reported for sonar problem 67
3.8 Sonar Signal Classifier: Cross Validation Parameters 68
3.9 Performance of Various Minimization Methods on Sonar Problem . . 69
3.10 Performance of Minimization Methods on Sonar with Weight Decay . 69
3.11 Cross Validation Parameters for Cascade Correlation Sonar Classifier 71
3.12 Cascade Correlation Sonar Signal Classifier Results 71

4.1 Minimization Algorithm Storage Requirements and Computation Types 76
4.2 Simulation Parameters 87
4.3 Classification accuracies (%) with weight decay 91
4.4 Average classification accuracies (%) with zero offsets 92

Chapter 1

Introduction

The Problem of Pattern Classification

There are many computational tasks, such as the recognition of various objects or signals, that are somehow handled very well by the human brain. This process of pattern classification allows one to easily categorize never-before-seen objects based on visual attributes such as shape and color, and recognize words spoken by people with different accents as being the same, based on certain information within the sound. Getting computers to perform pattern classification would be of great benefit in certain cases where many patterns such as postal codes must be classified one after another, creating very tedious work.

The Trouble with Traditional Computers

Computers were originally built to rapidly and reliably perform mathematical computation, but have trouble when it comes to performing the task of pattern classification. The inability of traditional computers to perform well on pattern classification problems is due to the amount of information that must be processed for classification to take place, and the inexact nature of classification problems. Most computers

are not equipped to efficiently handle large amounts of information presented all at once, and must follow certain procedures for classification supplied by programmers. These procedures may be difficult to set up and are not guaranteed to be the optimal ones. Quite often computers follow a sequential procedure when performing pattern classification — they will receive as input a vector of data (an "input pattern") corresponding to different aspects of an input picture, spoken word, etc. and will compare the input vector to each of many stored reference vectors one at a time. The input vector is classified as belonging to the same class as the "most similar" reference vector. This method of classification, called **template matching**, is used in some optical character recognition (OCR) systems that translate typed text into a form (binary) usable by a computer [2].

Template matching is an acceptable method of classification only if the reference vectors, the **exemplars**, are easy to set up. For typed text one need only include font libraries to use as the reference vectors. However, there are many more complex classification tasks, such as sorting mail by classifying the *handwritten* postal codes. For handwritten text, with all the different types of handwriting styles, setting up an adequate set of exemplars would require a great deal of time. A very large number would be required to get good classification accuracy [3]. When in use, the classification system would then have to traverse the very long list of exemplars one by one and determine which is most similar to the input. This would be very inefficient.

Rather than create a list of exemplars, one can design a set of mathematical operations to perform on the input data that extract certain important features of the data. The computer can then use the data produced by these operations, or **feature detectors** (for example edge finders), to aid in classifying the input data.

The feature detectors are difficult to set up, however, and there is never a guarantee that the optimal set of features will be extracted. It would be advantageous if the feature detectors could themselves be set up automatically by the computer.

The Promise of Neural Networks

Since the human brain handles the task of pattern classification so well, one would hope that **artificial neural networks**, computers with architectures that attempt to model the human nervous system, would be able to perform pattern classification as efficiently. Artificial neural networks (ANNs) have recently been applied to many pattern recognition problems (the types of problems studied in this thesis), in areas such as speech, computer vision, and medicine. Specific applications include learning to pronounce English text [63], detecting bombs in suitcases, and classifying lower back pain. ANNs have also been used for prediction (for example sunspot prediction [12]), pattern storage, and control (for example learning to follow a road [11]).

Artificial neural networks, in their many forms (see [4, 5, 6, 1] for overviews of some types), to date only model very little of the workings of the human brain. Nevertheless, there are two important features of real, biological, neural networks that most have. First, all ANNs have massively parallel computing architectures. They are capable of receiving, for example, an entire image or set of statistics about a signal as input, and producing an output in a few short processing steps. This parallelism, assuming the ANNs are implemented in parallel hardware, solves the problem that most traditional computers have with processing limited amounts of data at a time. A side effect of this massive parallelism is that neural networks are very fault tolerant. Failures in processors (the "neurons") or connections (the "synapses")

usually cause graceful degradation in performance rather than complete failure [7]. Second, most ANNs *learn* — they adapt their inner computations to better their classification accuracy during a learning, or training, process. This learning process frees a programmer from having to code extensive knowledge of a pattern classification problem. Learning also builds up fault tolerance. As a network's components fail, the rest of the network may be retrained and the network will learn to compensate for the failed components' behaviour (or lack thereof) [8].

There are two main types of artificial neural networks. A neural network is regarded as belonging to one of these two types depending on whether a "teaching signal" is fed into the network during training. A teaching signal is used to tell **supervised** artificial neural networks how well they are classifying each input pattern. The networks use the teaching signal to modify their inner computations so that hopefully they will perform better when the same input pattern is presented again to the network. Supervised ANNs include networks that learn through **reinforcement learning**. Here the networks are told whether their outputs are correct or incorrect only, but in general supervised neural nets receive a teaching signal that contains information about the degree of correctness. In contrast to supervised ANNs, **unsupervised** ANNs do not make use of a teaching signal; they are not given any information about whether their outputs are right or wrong. These networks are normally used for data clustering [9, page 343] rather than for pattern classification. Reinforcement and unsupervised networks are beyond the scope of this thesis, which will focus on pure supervised networks only.

By far the most popular method of training supervised ANNs is the Backpropagation of Errors (or just "backpropagation" or "backprop") training algorithm [1]. Here

a network computes the error in the output signal it has produced and propagates this error "backwards" through the network to modify its inner computation. Another popular training algorithm is the cascade correlation algorithm [10], which frees the network designer from worrying about how many processors to use by adding them one-by-one during learning.

Many cycles through a set of training data, or **training epochs** are usually required for a network to perform with an acceptable amount of accuracy. Once training is complete, the network will have built its own set of feature detectors with little human intervention, and will be able to classify input patterns.

1.1 Purpose

The purpose of this thesis is a study of the performance of backpropagation and cascade correlation neural network learning algorithms for learning real-world problems, with regards to resulting classification accuracy and amount of computation required. Various minimization (of output error) algorithms for use with either of the learning algorithms are included in the study. An analog hardware implementation of the basic backpropagation algorithm is also studied, because of the importance of implementing algorithms in dedicated hardware.

1.2 Problem

The backpropagation algorithm is very widely used for training supervised ANNs, but is very slow. As a result, there are very many neural network training (learning) algorithms currently being researched. Many of these are enhanced versions of the backpropagation algorithm, often nothing more than ordinary backpropagation with

a minimization algorithm added. There are three main problems usually present when an algorithm is introduced that cast some doubts on how well it actually performs. First, when a network is trained using a new algorithm, its classification accuracy is not compared against the accuracy of networks trained using other algorithms. Second, the classification problems chosen are much simpler and/or smaller than real-world type classification problems. Third, the inventors of new learning algorithms are not in agreement as to how to determine learning speed. This problem arises because there is no general agreement on when to stop training a neural network. A good methodology is needed to decide when to stop training.

In addition to deciding when to stop training a neural network, there are other factors that will determine the actual training time when the network is used in the real world (as opposed to the lab). Classification speed will be much faster if proper hardware is used for implementing the algorithm. Another factor affecting actual training time is the tedious process of training successively larger networks on the same problem to determine the number of processors required. The cascade correlation algorithm may alleviate this problem because it adds computing resources to a network automatically.

1.3 Scope

Chapter 2 introduces the algorithms; the first presented is backpropagation. Some pointers for using backpropagation effectively are then given. Next, minimization algorithms are presented that attempt to build on the basic backpropagation algorithm by computing and using additional information. Finally, the cascade correlation algorithm is presented. Algorithms mentioned in Chapter 2 that use excessive

mathematics are presented more fully in Appendix A.

In Chapter 3, the various training algorithms are analysed, using various classification problems, with regards to training time and classification accuracy. The problems include a real-world handwritten character recognition problem, one of the problems most intensively researched by neural network researchers today. A discussion of performance analysis methods is also presented in Chapter 3.

In Section 4.1 the algorithms are analysed with regards to hardware compatibility. Some factors influencing hardware compatibility include types of computations required (some computations are more difficult to implement than others), and locality of the operands in the computations. This section forms a bridge to Section 4.2 where a much more rigourous analysis of the basic backpropagation algorithm is presented with regards to hardware compatibility. Here it is assumed that the entire algorithm is implemented in analog hardware. It is discovered how well the algorithm can perform assuming various analog hardware properties that will make the computations deviate from the mathematically ideal ones.

Chapter 2

Background

2.1 Typical Network Architecture and Operation

Artificial neural networks all consist of processors (the "neurons", "processing units", or just "units") and connections between the processors (the "synapses" or "links"). A very common network architecture for pattern classification is the **multi-layered perceptron** (MLP). Here neurons are arranged in layers, with full connectivity between layers. The inputs are not neurons, and do not count as a layer. Neurons between the input and outputs are called **hidden** neurons. The left-hand side of Figure 2.1 shows such a network.

A common way of using the output neurons of a MLP is to have one output for each class of input, for example "apple","orange","pear". When an input pattern is presented to the network, hopefully only the neuron corresponding to the correct class will become active (produce a strong output signal) and the rest will remain inactive. Normally the neuron corresponding to the correct class is only required to be more active than the rest; there are methods of enhancing the output of the most active neuron to ease the classification process (see [4] for an example). By using feedback connections within a network, and using some neurons to store partial results, one

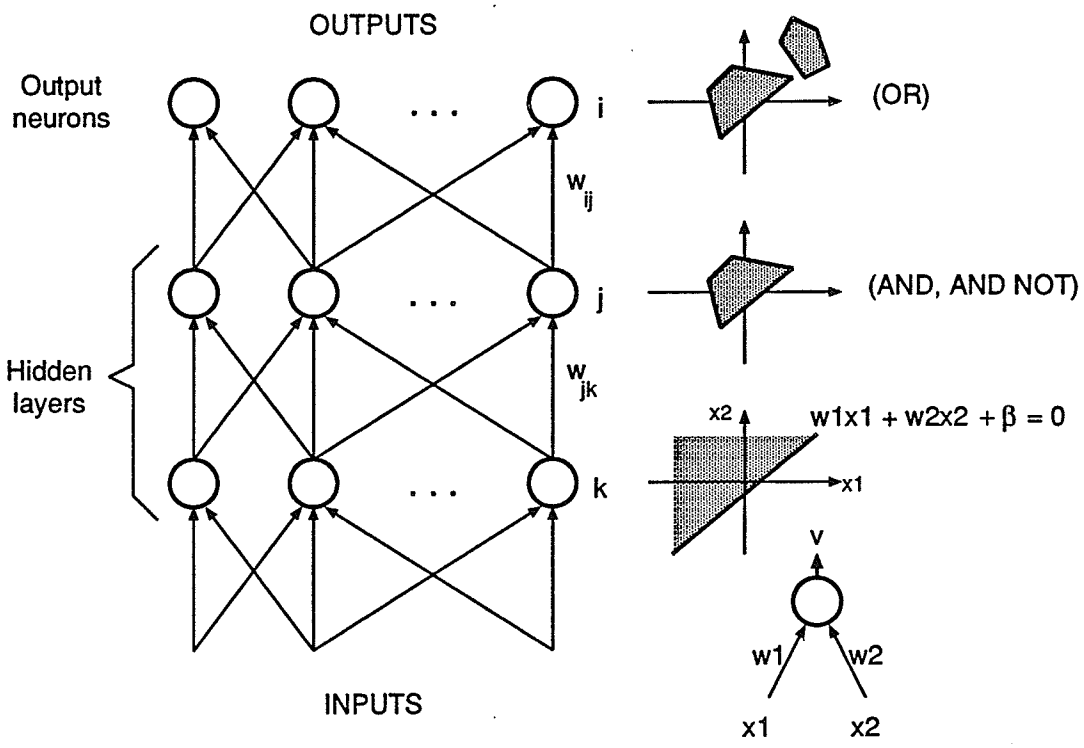

Figure 2.1: Multi-layered Perceptron with Decision Regions

can use a network to do things like complete a sequence of input patterns [1]. These **recurrent networks** will not be pursued in this thesis.

Typically each neuron within a network will take a weighted sum of its inputs, add a **bias**[1], and pass the result through a nonlinear function, known as the **activation function** as shown in Figure 2.2. Two common nonlinear functions are versions of **sigmoid** functions, functions with an S-shaped characteristic. These are the logistic function pictured in Figure 2.3, and the hyperbolic tangent (tanh) function.

The bias term is identical to the weight of a connection projecting from a neuron with output 1. The need for biases will be made clear shortly.

It turns out that to perform any mapping between input and output, one only need use a three-layered perceptron, that is, one with two hidden layers and one out-

[1]The bias can be thought of as the negative of a threshold.

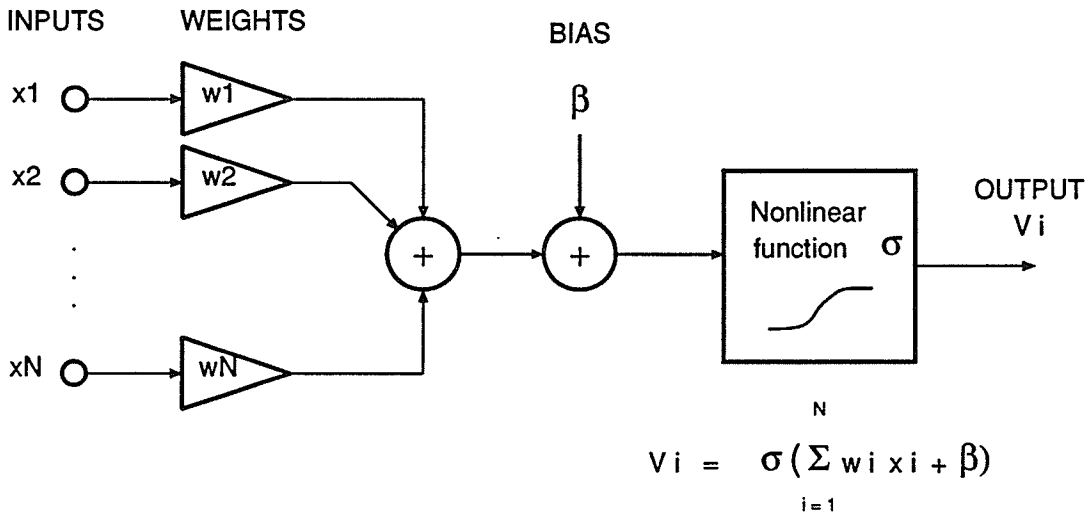

Figure 2.2: Typical Neuron Function

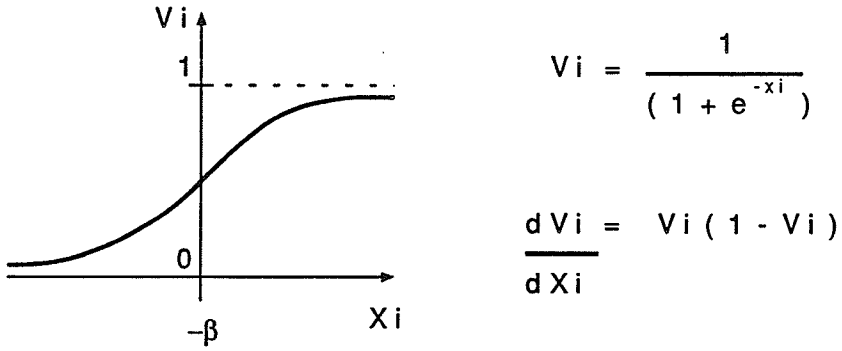

Figure 2.3: The Logistic Function

put layer [4],[6, p.142]. Assuming the neurons use step functions as their nonlinear functions, neurons in the first layer (directly above the inputs) can solve only classification problems that are linearly separable, that is, where the classes are separated by hyperplanes in input space (lines if there are only two inputs). The neurons' biases allow these hyperplanes to avoid passing through the origin in input space, allowing for greater flexibility. The right-hand side of Figure 2.1 shows **decision regions** possible at various layers of a network, assuming two inputs x1 and x2, and two classes. Inputs falling within the shaded decision regions will be classified by the network as class 1, otherwise they will be classified as class 2. Neurons in a second layer can take the logical AND of neuron outputs from the first layer to form *convex* decision regions. A convex decision region is one in which a line drawn between any two points on the edge of the decision region must pass entirely within the decision region. Neurons in a third layer can take the logical OR of neuron outputs from the second layer and combine the convex decision regions to form arbitrarily complex decision regions. With these arbitrarily complex decision regions a neural network can theoretically achieve 100% accuracy on a particular classification problem, if enough neurons are used. The more commonly used sigmoid nonlinear functions "blur" the decision boundaries slightly, but this discussion is still valid. Note that neurons with linear transfer functions would not be very useful in a layered network, as all the functions computable by more than one layer of neurons could be computed by a single layer of neurons. A linear function of the inputs would be the result, no matter how many layers of neurons are used.

2.2 The Backpropagation Learning Algorithm

Widespread use of the backpropagation learning algorithm for neural networks has only occurred within the past few years. The algorithm became popular when introduced by Rumelhart, Hinton, and Williams in 1986 in the classic books *Parallel Distributed Processing* [1], but had been invented independently by several other authors, as early as 1969 [13, 14, 15]. The learning algorithm will work on a multi-layered perceptron with any number of layers, and also on networks with shortcut connections spanning more than one layer. This algorithm was a significant breakthrough because up until it was discovered, it was not known how to train networks with hidden neurons. Networks that could be trained were restricted to the single-layered variety, which as already shown can only solve linearly separable problems. A small problem such as exclusive-OR could not be solved. Minsky and Papert, in their famous book *Perceptrons*, did not believe that any learning algorithm for a multi-layered neural network could be found [16]. This conclusion resulted in a very quiet period in neural network research from the time the book was published in 1969 to 1986 when backpropagation was made popular. Rumelhart et. al's success in achieving popularity for backpropagation is partly due to the fact that they showed that the algorithm could solve a number of problems, such as parity (generalized exclusive-OR), detection of symmetry, and simple vision problems.

Normally when one trains a MLP, one tries to minimize half the *squared output error* over all training patterns:

$$E = \frac{1}{2} \sum_{patterns} \sum_{outputs\ i} (V_i - V_i^d)^2, \tag{2.1}$$

where V_i and V_i^d are the actual and desired **activations** (outputs) respectively of

output neuron i. The $1/2$ is included to simplify the derivative, which will be needed in the learning calculations. The desired activation is known as the **target** activation. The target is normally set to the upper limit of the neuron activation function if the neuron is to become active when the input pattern is presented, otherwise it is set to the lower limit.

If one thinks of the classification problem as a landscape with the error measure E the height and the weights the coordinates, backpropagation finds the gradient, or direction of steepest ascent, of this landscape at the current position (determined by the weights). The algorithm does this by finding for each weight of a connection between two neurons i and j the partial derivative of error with respect to that weight $(\partial E/\partial w_{ij})$. If no enhancements are made to the backpropagation method, then after each gradient calculation an attempt is made to go downhill in the error landscape by changing each weight by an amount proportional to the negative of its contribution to the gradient. That is, for a weight of connection between neurons i and j,

$$\Delta w_{ij} = -\epsilon \frac{\partial E}{\partial w_{ij}} \tag{2.2}$$

where ϵ is the **learning rate.**

Each component of the gradient, or **weight error derivative** $\partial E/\partial w_{ij}$ is found using the chain rule. The following derivation will show how the changes for weights w_{ij} and w_{jk} in the network of Figure 2.1 are calculated.

In general, for the weight of a connection between neurons a and b, where a is the neuron "closer" to the output,

$$\frac{\partial E}{\partial w_{ab}} = \frac{\partial E}{\partial V_a} \cdot \frac{\partial V_a}{\partial X_a} \cdot \frac{\partial X_a}{\partial w_{ab}} \tag{2.3}$$

where $V_a \equiv$ neuron a's output, $X_a \equiv$ weighted sum including bias (sometimes called the **net input** to neuron a).

$\partial E/\partial V_a$:

for output neuron $i, \partial E/\partial V_i = V_i - V_i^d$

for hidden neuron $j, \partial E/\partial V_j = \sum_o(\frac{\partial E}{\partial V_o} \cdot \frac{\partial V_o}{\partial X_o} \cdot \frac{\partial X_o}{\partial V_j}) =$

$\sum_o((V_o - V_o^d) \cdot \frac{\partial V_o}{\partial X_o} \cdot w_{oj})$,

where o is an output neuron.

$\partial V_a/\partial X_a$: This term depends on the type of nonlinearity used by the neurons. Since we are taking the derivative of the neuron's nonlinear function, this function must be differentiable.

If the logistic function is used,

$\partial V_a/\partial X_a = V_a(1 - V_a)$

If the tanh function is used,

$\partial V_a/\partial X_a = 1 - V_a^2$.

Both of these derivatives are very simple mathematically, and therefore easy to implement in hardware.

$\partial X_a/\partial W_{ab} = V_b$

Given the above, the following equation for Δw_{ij} can be constructed and applied to any input connection of an output neuron:

$$\Delta w_{ij} = -\epsilon\frac{\partial E}{\partial w_{ij}} = \epsilon f'(net_i)V_j(V_i^d - V_i) \tag{2.4}$$

and the following equation for Δw_{jk} can be constructed and applied to any input connection of a hidden neuron:

$$\Delta w_{jk} = -\epsilon\frac{\partial E}{\partial w_{jk}} = \epsilon f'(net_j)V_k\sum_u(\frac{\partial E}{\partial V_u} \cdot f'(net_u) \cdot w_{uj}) \tag{2.5}$$

where $f'(net_*) = \partial V_*/\partial X_*$, and u is a neuron in the next "higher" layer (closer to the output) than neuron j. In this example, neurons u are in fact output neurons, but they need not be in general for equation 2.5 to hold.

It should be noted that other than for the non-linear function, all network operation is controlled by multiplication and addition operations. The summation in equation 2.5 contains values that must be passed back only from the adjacent higher layer in the network. Each synapse has easy access to the rest of the operands required; they are the outputs of the neurons at either end of the synapse, and (for output neurons only) a target output presented directly to the neuron. Therefore, all operands in the learning equations are available locally.

It should also be noted that the error landscape for a multi-layered perceptron will contain **local minima** with gradient zero and solution non-optimal (the optimal solution is the **global minimum**). It is very possible that backpropagation will make the network settle into a non-optimal solution. However, the non-optimal solutions are said to usually be almost as good as the one at the global minimum.

2.3 When to update the weights

The simple weight update rule of equation 2.2 can be applied after each training pattern has been presented to the network. In this case one is said to be using **on-line [weight] updating**. It is also possible to accumulate the weight change calculations over a number of training patterns (often the entire training set) before actually making the weight changes. This technique is known as **batching** or **batch [weight] updating**. The latter method results in a better approximation to the true gradient direction for the classification problem. However, much more time is required between weight changes and there is no guarantee that batching will eventually lead to a significantly better solution than on-line updating.

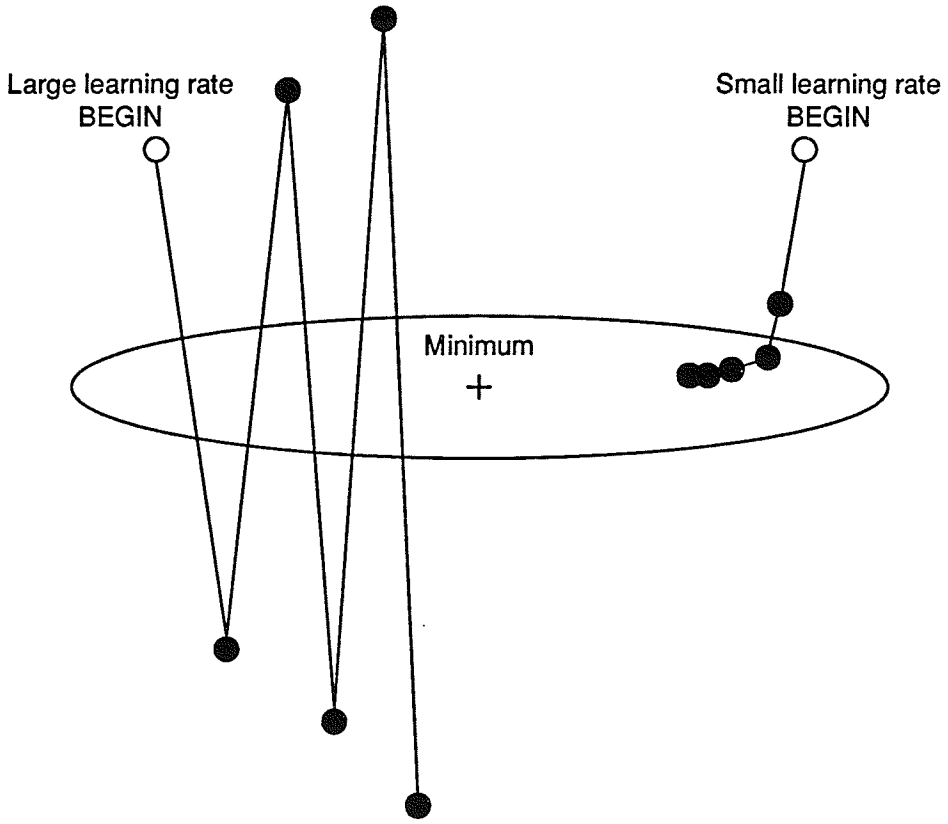

Figure 2.4: The Problem with Large Learning Rates

2.4 Problems with backpropagation, and fixes

The biggest problem with the backpropagation learning algorithm is its slowness. Even a simple problem like exclusive-OR may require hundreds of passes through the training dataset before an acceptable solution is found. Since the simple learning algorithm does not attempt to determine an optimal value for the learning rate, the latter must be set very small to avoid ending up worse off than before in terms of classification error. It is very possible for the network's weights to oscillate without converging if the learning rate is set too large, as shown in Figure 2.4. In the figure, the error landscape is a sort of "ravine".

In addition to this "ravine" problem, there are other problems with backpropaga-

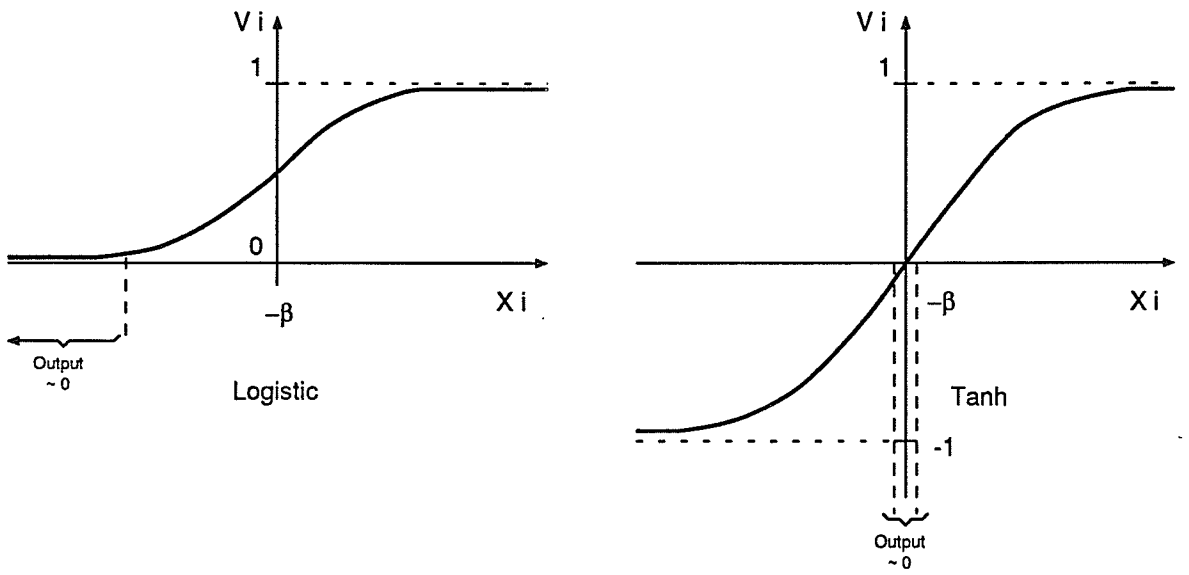

Figure 2.5: Logistic / Tanh Function Comparison

tion that make it very slow to learn. If a neuron's output is zero for a particular input pattern, then the weights of connections projecting from this neuron won't be able to change. An easy solution to this problem is to use tanh functions instead of logistic ones for the neurons' nonlinear transter functions. As illustrated in Figure 2.5, if the logistic nonlinearity is used, then any negative net input of large magnitude will produce a neuron output close to zero. The neuron output will in fact go less than 0.01 for any net input lower than about -4.5. If the tanh nonlinearity is used, the magnitude of neuron output will go less than 0.01 only if the net input is within the small range -0.01 and +0.01.

Another problem with simple backpropagation is that, for both the logistic and tanh activation functions, when the magnitude of the net input to the neuron is very high, the derivative of the activation function ($f'(net_i)$) will go to zero. All calculated changes for weights feeding into the neuron will then go to zero as well, regardless of whether or not the neuron's output is close to the optimal one. These weights will

not change easily and will not contribute to the learning of the classification problem. This problem may be a fatal one to machines with limited precision, where small values will be truncated. A solution to this problem is to initialize the weights to values inversely proportional to the neuron fan-in (assuming that the hardware will allow for small enough weights). This way there is less likelihood that the neuron will "saturate". Another preventative measure against saturating sigmoids is to set the target values of the output neurons to be values other than the sigmoid limits. For example, for output neurons with tanh sigmoids, target values of -0.7 and +0.7, instead of -1 and +1, have been suggested [17]. Some other preventative measures have been suggested in [61], but these modify the backpropagation algorithm itself so that it no longer performs true gradient descent, and were only shown to work on encoder problems (see page 31).

A possible solution to the saturation problem that also attempts to alleviate any hardware problems due to finite weight range in hardware is **weight decay**, where weights decay to zero over time. This idea is discussed further in Section 2.7.

2.5 Backpropagation Enhancements

In addition to the improvements to the gradient calculation itself, there have been very many proposals to speed up backpropagation by trying to make intelligent use of the gradient and/or other information. A discussion of all of these would fill several volumes. However, neural network research has progressed to the point where several have emerged as significant advances and are now available in commercial and public domain neural network simulators. A discussion of them would therefore be beneficial.

Instead of using a fixed learning rate ϵ one may use a line search in the direction of

the gradient to determine an optimal learning rate to use when updating the weights. In other words, one chooses λ to minimize the error in

$$\mathbf{x} = \mathbf{x}_0 + \lambda \mathbf{d} \tag{2.6}$$

where $\mathbf{x}$ is the new weight vector, $\mathbf{x}_0$ is the old weight vector (before the weight change), and $\mathbf{d}$ is the direction in which to travel. When d is chosen to be $-\nabla E(\mathbf{x}_0)$, the resulting method is known as true **steepest descent.** The simplest line search is to repeat a fixed step until the error no longer decreases [18]. Another line search, sometimes called **Ray's** line search [20], works as shown in Appendix A. Note that for a line search to work properly, the calculated gradients at nearby points along the search direction must be similar. Batch updating must therefore be used.

True steepest descent is said to provide a speedup over ordinary backpropagation, but the network sometimes fails to find a good solution at all [22]. When using steepest descent, succesive steps are perpendicular because

$$0 = \frac{\partial}{\partial \lambda} E(\mathbf{x}_0 - \lambda \nabla E(\mathbf{x}_0)) = \nabla E(\mathbf{x}_0) \cdot \nabla E(\mathbf{x}) = \nabla E^{old} \cdot \nabla E^{new} \tag{2.7}$$

Therefore the approach to the minimum is always a zigzag path [6, p.126].

A better method is to use as the new search direction for the line search some compromise between the newly calculated gradient direction and the old search direction:

$$\mathbf{d}^{new} = -\nabla E^{new} + \beta \mathbf{d}^{old} \tag{2.8}$$

for some appropriate β. Doing this, of course, requires a good approximation to the true gradient direction for the classification problem, meaning batch updating.

There are two well-known algorithms that use this "compromise" method — one very simple and one more complicated. The simple one is referred to as **momen-**

tum [1] and is so widely used that many mistake it as being part of the standard backpropagation algorithm. The idea is, at every weight, to make the update

$$\Delta w_{ij}(t) = -\epsilon \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1). \tag{2.9}$$

α, like ϵ, is a global parameter uniform over all weights in the network. The momentum method may actually cause the weights to move in an uphill direction in the error landscape, so it is not normally used with a line search. Nevertheless it does usually give a significant speedup over plain backpropagation in many cases (as will be shown later). The speedup is still limited, however, because if all derivatives of a weight over time are assumed equal to one, then Δw_{ij} will converge to $\frac{-\epsilon}{1-\alpha} \frac{\partial E}{\partial w_{ij}}$ [23].

A more complex way of making a compromise between the current and previous search directions is utilized by **conjugate gradient** methods. Here the new search direction is chosen so that each new search direction spoils as little as possible the minimization achieved by the previous one [6, p.126]. This means that we need

$$\mathbf{d}^{old} \cdot \nabla E(\mathbf{x}_0 + \lambda \mathbf{d}^{new}) = 0 \tag{2.10}$$

which is the same as the condition

$$\mathbf{d}^{old} \cdot \mathbf{H} \cdot \mathbf{d}^{new} = 0. \tag{2.11}$$

The vectors $\mathbf{d}^{old}$ and $\mathbf{d}^{new}$ are then said to be *conjugate.* $\mathbf{H}$ is the Hessian matrix, the nxn matrix of second derivatives of the n weights in the network with respect to the output error [$\mathbf{H}_{ij} = \partial^2 E/(\partial w_{ab})(\partial w_{cd})$]. Storing this matrix would be quite a formidable task when there are thousands of weights in a network. Fortunately, β in equation 2.8 can be found such that equation 2.11 is satisfied without knowledge of $\mathbf{H}$. According to the Polak-Ribiere rule [6, p.126],

$$\beta = \frac{(\nabla E^{new} - \nabla E^{old}) \cdot \nabla E^{new}}{(\nabla E^{old})^2} \tag{2.12}$$

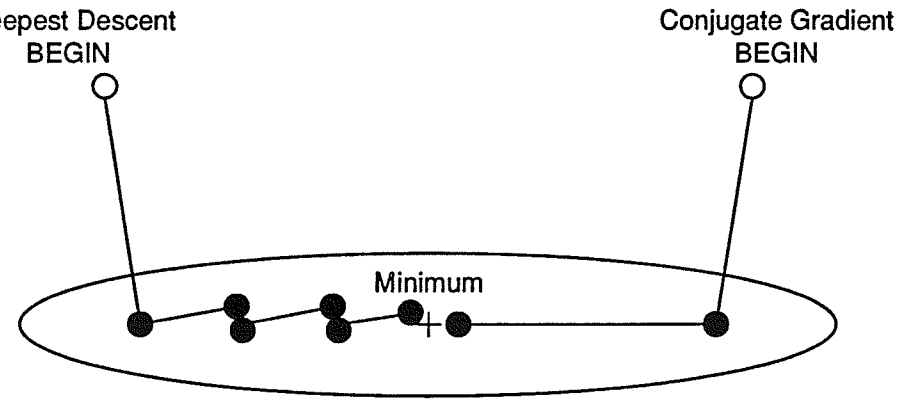

Figure 2.6: Steepest Descent / Conjugate Gradient Learning Comparison

A comparison of Polak-Ribiere conjugate gradient and true steepest descent on a simple quadratic problem (the optimal type of problem for conjugate gradient) is shown in Figure 2.6.

Another rule for finding β, called **Rudi's conjugate gradient** [20], is being used by Geoffrey Hinton's group. This rule is shown in Appendix A. It requires more computation per weight change but is claimed to give faster convergence to a good solution.

Another well-known enhancement to standard backpropagation is the **delta-bar-delta** rule, which was introduced by Robert Jacobs [23]. Delta-bar-delta does not follow as mathematically sound a procedure as conjugate gradient methods, but instead makes use of the following heuristics:

1. Every parameter (weight) of the performance measure to be minimized (the output error) has its own learning rate.

2. Each learning rate varies over time.

3. When the derivative of the error with respect to a weight possesses the same sign for several consecutive time steps, the learning rate for that weight is increased.

4. When the sign of the derivative in 3 alternates for several consecutuve time steps, the learning rate for that weight is decreased.

Mathematically, the learning rate update rule is defined as follows:

$$\Delta\epsilon_{ij}(t) = \begin{cases} \kappa & \text{if } \overline{\delta_{ij}}(t-1)\delta_{ij}(t) > 0 \\ -\phi\epsilon_{ij}(t-1) & \text{if } \overline{\delta_{ij}}(t-1)\delta_{ij}(t) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

where $\delta_{ij}(t) = \partial E/\partial w_{ij}$, and

$$\overline{\delta_{ij}}(t) = (1-\theta)\delta_{ij}(t) + \theta\,\overline{\delta_{ij}}(t-1). \tag{2.14}$$

The vector of $\overline{\delta_{ij}}$ terms is sometimes thought of as a momentum vector, and equation 2.14 is replaced by the momentum equation [17]:

$$\overline{\delta_{ij}}(t) = \delta_{ij}(t) + \alpha\,\overline{\delta_{ij}}(t-1). \tag{2.15}$$

According to this learning rule, when the current derivative of a weight and an exponentially weighted average of the weights's previous derivatives have the same sign, the learning rate is incremented by an additive amount. If they have opposite signs, the learning rate is decremented by a multiplicative amount. The algorithm may appear much more difficult to use than other enhanced backpropagation algorithms because of the extra parameters, but it is actually reasonable because networks are quite insensitive to changes in the parameters κ and ϕ [17]. Values of 0.1 for κ and 0.9 for ϕ have been suggested to keep ϵ_{ij} around 1.0 when δ_{ij} changes randomly [20], and a momentum value α of 0.9 is generally agreed upon. Delta-bar-delta would not appear to work very well with on-line updating because the sudden shifts in gradient direction estimation from pattern to pattern would make the learning rate very quickly go to zero. It is therefore advised to use batch updating.

So far all of the methods listed above use only first-order information about the error landscape. If we knew about the second order information we could make much better weight changes. This statement is borne out mathematically by noting that the error function can be approximated by a Taylor series expansion about the current point in weight space (i. e. the current weight configuration) $\mathbf{x}_0$ as

$$\mathbf{E}(\mathbf{x}) = E_0 + (\mathbf{x} - \mathbf{x_0}) \cdot \nabla E(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x_0}) \cdot \mathbf{H} \cdot (\mathbf{x} - \mathbf{x_0}) \tag{2.16}$$

The rest of the terms in the expansion are neglected. The optimal weight change vector is obtained by setting the derivative of the preceding expression to zero. Here the error will be at a minimum.

$$0 = \nabla E(\mathbf{x}_0) + \mathbf{H}(\mathbf{x} - \mathbf{x_0}) => \Delta \mathbf{w} = -\mathbf{H}^{-1}\nabla \mathbf{E}. \tag{2.17}$$

This is nothing other than **Newton's method.** In one dimension it reduces to

$$x = x_0 - \frac{E'(x_0)}{E''(x_0)} \tag{2.18}$$

Newton's method is impractical because it requires inversion of an nxn matrix before each weight change, requiring order n^3 steps each time [6, p.125]. One group proposed that the off-diagonal terms of the Hessian be omitted [24]. When this is done each weight change reduces to

$$\Delta w_{ij} = -\frac{\partial E}{\partial w_{ij}} / \frac{\partial^2 E}{\partial w_{ij}^2} \tag{2.19}$$

This new update rule is called the **pseudo-Newton rule.** The $\partial^2 E / \partial w_{ij}^2$ term is expanded in Appendix A. It is huge, especially for weights of incoming connections of hidden neurons. It also requires external (to the network) storage of partial computation if more than two layers of neurons are used. One advantage of the pseudo-Newton

rule is that it requires no learning parameters, save a parameter that can be added to the denominator to ensure it is always positive [6, p.128]. Equation 2.19 makes intuitive sense because as the second derivatives decrease, a plateau in error space is being reached and the weight changes should become larger. This method can of course be used with batch updating, but has also been used with on-line updating [25]. It is not, however, known to provide a significant advantage in speed over simple backpropagation [6, p.128].

The last enhancement to backpropagation that will be presented here is one that uses approximations to the second derivatives of the error function. The **quickprop** algorithm, proposed by Scott Fahlman [61] normally updates each weight according to

$$\Delta w_{ij}(t) = -\epsilon S(t) + \frac{S(t)}{S(t-1) - S(t)} \Delta w_{ij}(t-1) \qquad (2.20)$$

where $S(t)$ and $S(t-1)$ are the current and previous values of $\partial E/\partial w_{ij}$, and ϵ controls the "steepest descent" component of the learning equation (it may be thought of in the same way as the learning rate for standard backprop). Modified versions of equation 2.20 are sometimes applied depending on the signs and magnitudes of $S(t)$ and $S(t-1)$. The full quickprop algorithm is presented in Appendix A.

Fahlman suggests using weight decay and batch updating with quickprop.

2.6 The Cascade Correlation Algorithm

The Cascade Correlation (or just "cascor") algorithm was developed by Scott Fahlman and Christian Lebiere. As mentioned earlier, the algorithm works by adding neurons to a network one by one during learning, thereby automating the process of training increasingly larger networks. The algorithm was not developed for this purpose, but

instead to attempt to solve a problem with backpropagation not yet mentioned: the **moving target problem.** This problem was very well described in the cascade correlation seminal paper [10]:

> Because all of the weights in the network are changing at once, each hidden unit sees a constantly changing environment [meaning constantly changing goals, or targets]. Instead of moving quickly to assume useful roles in the overall problem solution, the hidden units engage in a complex dance with much wasted motion.

The way that cascade correlation attempts to overcome the moving target problem is to have each hidden unit learn a set of weights then freeze them forever. The algorithm works as shown in the flow diagram of Figure 2.7.

Use of more than one "candidate" unit will reduce the chance that a neuron that gets "stuck" at a poor covariance will be added to the network. The covariances between output error and candidate unit activation form a new performance measure that must be used when training the candidate units. Define S as

$$S = \sum_o |\sum_p (V_p - \overline{V})(E_{p,o} - \overline{E_o})| \tag{2.21}$$

where o is an output unit, p is a training pattern, V is the output of the candidate unit, and E_o is the error observed at output unit o. $\overline{V}$ and $\overline{E_o}$ are the values of V and E_o averaged over all training patterns. This function can be computed using a "shortcut formula" [27]:

$$S = \sum_o |\sum_p (V_p E_{p,o} - \overline{V} \sum_{p,o} E_{p,o})| \tag{2.22}$$

We wish to maximize S, so we compute $\partial S / \partial w_{ij}$, the partial derivative of S with respect to each of the weights w_{ij} of each of the candidate unit i's incoming

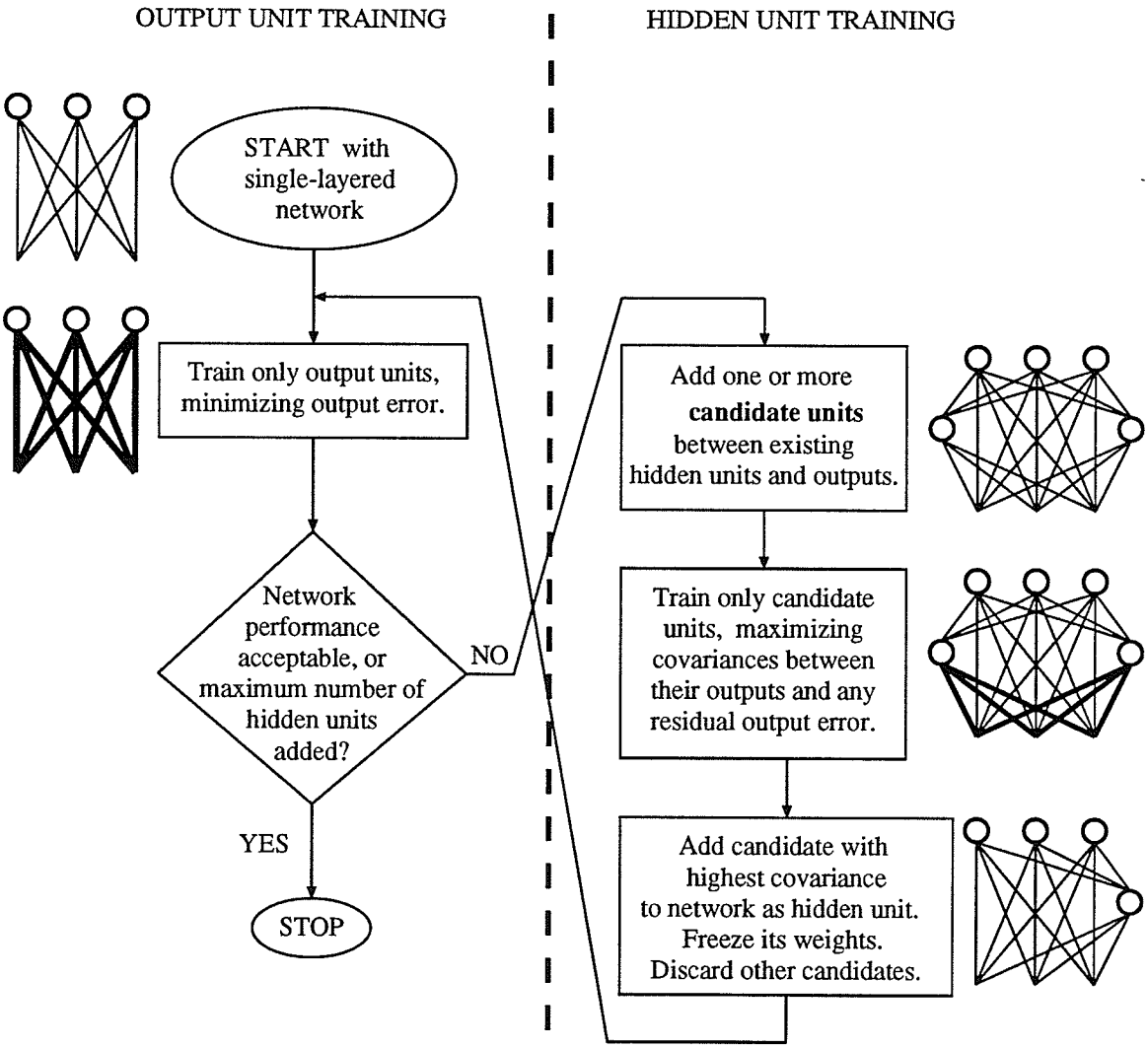

Figure 2.7: The Cascade Correlation Algorithm

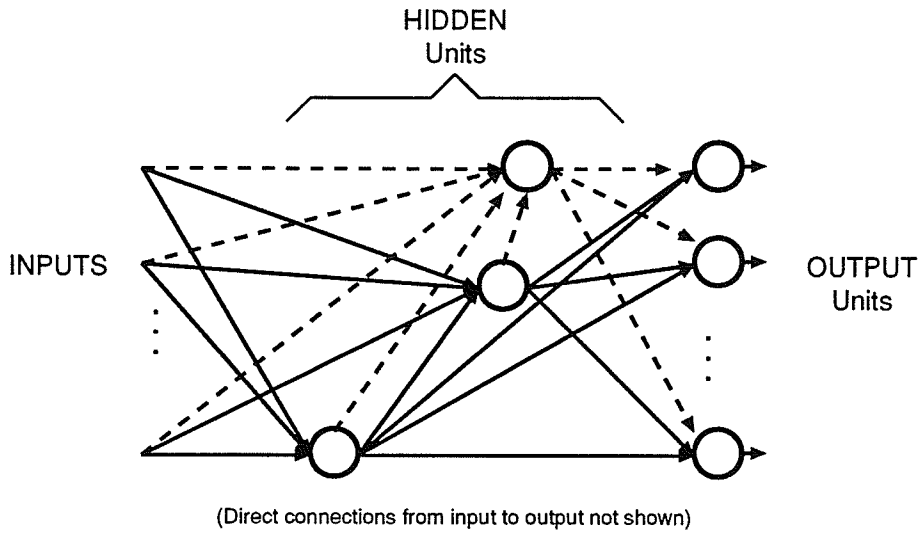

Figure 2.8: A Typical Network Constructed using Cascade Correlation

connections:

$$\frac{\partial S}{\partial w_{ij}} = \frac{\partial S}{\partial V_i} \cdot \frac{\partial V_i}{\partial X_i} \cdot \frac{\partial X_i}{\partial w_{ij}} = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E_o}) f'_p V_j \tag{2.23}$$

where σ_o is the sign of the expression within the absolute value sign in equation 2.21, f'_p is the derivative for pattern p of the candidate unit's activation function with respect to the sum of its inputs, and V_j is the input the candidate receives from input or hidden unit j for pattern p.

Each new hidden unit can connect to each previous hidden unit. This results in networks that resemble the one in Figure 2.8. Since each hidden unit performs a nonlinear function of all previous hidden units as well as the inputs, the result is potentially more powerful networks than MLPs, where the connections between hidden units are not present.

Now that the gradients in Cascade Correlation networks have been defined (components are $\partial E/\partial w_{ij}$ for output unit training and $\partial S/\partial w_{ij}$ for candidate unit training), any minimization algorithms that use the gradient information (i. e. momentum,

quickprop, conjugate gradient, etc) can be used to hopefully speed learning. The candidate units must be trained using batch updating, so that average output unit errors and average candidate outputs can be calculated.

2.7 Generalization

With most pattern classification systems it is desired to have the system correctly classify patterns that were not used when the classification system was designed. With neural networks, this means that the network should be able to correctly classify patterns that were not part of the training dataset. The ability of the network to do this is called its **generalization** ability. If a network is presented with only a few training patterns it may try to "memorize" them and will do poorly when presented with new data, assuming that the training patterns did not form a good representation of the classification problem. A general guideline is to use a larger number of examples in the training dataset than there are weights in the network. This will allow the training data to "affect" all the weights and get rid of any initial conditions (weight values) in the network that will interfere with learned information and hamper generalization [28]. Guidelines on the number of training patterns to achieve a *certain level* of generalization have been proposed by Baum and Haussler [29] among others. However, these are estimations for general classes of problems rather than particular problems and tend to overestimate the number of training patterns required for a particular problem [28].

Using weight decay is said to improve generalization [62]. Instead of minimizing just the squared error, we minimize

$$C = \frac{1}{2} \sum_{i=1}^{\#outputs} (V_i - V_i^d)^2 + \lambda/2 \sum_{j=1}^{\#weights} w_j^2 \qquad (2.24)$$

Therefore each weight is updated (assuming simple backpropagation) according to

$$\Delta w_j = -\epsilon(\frac{\partial E}{\partial w_j} + \lambda\, w_j) \tag{2.25}$$

At equilibrium $\partial C/\partial w_j = 0$, and

$$-\frac{\partial E}{\partial w_j} = \lambda w_j \tag{2.26}$$

What equation 2.26 means is that at an advanced stage of learning, large weights are guaranteed to be doing useful work in reducing the error rather than just "hanging around" forcing sigmoids to saturate and making the network harder to train. Weights that do little useful work in reducing the error will decay to zero. A weight decay method should make it easier to strip away any excess connections (and possibly entire neurons) from the network after training. However, in hardware excess neurons may add fault tolerance to the network, as there is more to retrain when failures occur, so the network designer should also keep this in mind.

Another factor affecting generalization is the learning time. When ANNs have been trained too long they may start to "over-fit" the training data. Therefore, rather than worrying about the number of parameters in the network, the network designer could use cross-validation to decide when to stop training. The cross-validation approach generally used for neural networks is to present them with patterns not part of the training set (ie: part of the test set) from time to time. When the error on the test set starts to increase, then it is time to stop training the network. Possible cross-validation methods are presented in [31] and [32]. No one method of performing cross-validation has been agreed upon.

Chapter 3

Performance Analysis

The backpropagation and cascade correlation learning algorithms, and several minimization algorithms, have been introduced. We will now see how well these algorithms perform on pattern classification problems.

3.1 Methods for Performance Analysis

As has already been mentioned, different authors use different performance measures when studying neural network learning algorithms. The differences in performance measures arise because of different ways of determining that learning is complete, and different ways of taking "failed" learning trials into account when averaging learning times for a group of trials. This section will outline several possible performance measures, and will give reasons for selecting the ones adopted for this work.

There are two types of classification problems that can be learned by a supervised neural network, and each requires separate performance measures. These are:

- **Type 1**. Problems with a finite amount of possible training patterns, and the possibility that all training patterns can be correctly classified. The input data are usually binary. One problem of this type is the "parity" problem, where

the network must correctly determine whether the (binary) input pattern has even or odd parity. Another type 1 problem is the "encoder" problem , where the network is presented with a "1 of n" binary input pattern and an identical target pattern. The network is given a smaller number of hidden units than the inputs, and must learn to encode the input information as the activations (outputs) of this small number of hidden units. Generalization is not an issue with type 1 problems.

- **Type 2.** Problems with an infinite number of possible training patterns and/or no chance of correct classification of all training patterns. The input data are normally analog. Generalization becomes an important issue with type 2 problems. Some available data are normally set aside as test data, used for checking generalization performance.

3.1.1 Methods for Type 1 Problems

Possibly the most thorough discussion of possible benchmarking methods for type 1 problems is given by Scott Fahlman [61]. For problems with binary targets[1] Fahlman favors determining that learning is complete when all outputs are "correct", correctness determined according to the following: If the target is low (near 0 for logistic neurons, -1 for tanh) and the output is in the bottom 40% of its range, or if the target is high (near 1) and the output is in the upper 40% of its range, then the output is correct. If the output is within the middle 20% of its range, then the output value is "marginal" and is not counted as correct. This is because small noise applied at the input signal could too easily result in the output going into the wrong half of its

[1] Target values near the extremes of the output neurons' ranges. When each output represents one class this is common.

range.

The **Xerion** neural network simulator [19], used for the simulations in this thesis, does not use Fahlman's criterion to stop training, but instead uses the following criteria:

1. Error measure < threshold1, or

2. (|gradient vector| / |weight vector|) < threshold2

The error measure in the first criterion can be anything the user chooses; commonly it is the squared output error (equation 2.1). The second criterion is known as **convergence.**

It is the opinion of this author that the first criterion shown above is acceptable for type 1 problems. Fahlman, on the other hand, considers the criterion unnecessarily strict because, as he puts it, "some algorithms may produce useful outputs quickly, but take much longer to adjust the output values to within the specified tolerances." This is of course true, however I feel that the extra time to get the error below the threshold may not be unneccessary; the network will exhibit greater robustness to noisy inputs if it is trained to produce outputs very near the targets instead of just to within 40%. Also, when one is studying the *relative* effectivenesses of algorithms instead of determining that "algorithm x can learn problem y in z training epochs", then simply waiting for the error measure to get below a threshold is not unreasonable. There is the possibility that two learning algorithms will produce correct outputs according to Fahlman's criterion at about the same time, but one will have a harder time getting the error measure to below the threshold. Instead of reporting these learning algorithms as similar in speed, the algorithm that allowed the network to find a more robust solution in a shorter period of time should be considered the faster.

Of course, the criterion of error being below threshold will not always be met. If it is not met, the learning trial can be considered a failure and reported separately from the successful trials [23], or the trial can be restarted with a new set of weights and learning time accumulated [61]. The first method is the one used here. The average in the second method hides the number of failures, possibly something the network designer would like to know, and the following shows that it can be computed even if the first reporting method is used:

Theorem 3.1

Let P_f = the probability of failure given a certain learning algorithm, $\overline{S}$ = the average learning time (epochs, number of pattern presentations, etc.) on the successful trials, and W = the learning time wasted during a failed attempt. Then the average learning time with restarting can be approximated by

$$\overline{Time} = \frac{(1 - P_f)\,\overline{S} + P_f\,W}{1 - P_f} \tag{3.1}$$

Proof:

$$\begin{aligned}
\overline{Time} &= (1 - P_f)\overline{S} + P_f(W + (1 - P_f)\overline{S} + P_f(W + (1 - P_f)\overline{S} + P_f(W + \ldots))) \\
&= (1 - P_f)\overline{S} + P_f\,W + P_f\,(1 - P_f)\overline{S} + P_f\,P_f\,W + P_f\,P_f\,(1 - P_f)\overline{S} + \ldots \\
&= \sum_{i=0}^{\infty} P_f^i((1 - P_f)\overline{S} + P_f\,W)) \\
&= \frac{(1 - P_f)\overline{S} + P_f\,W}{(1 - P_f)}
\end{aligned}$$

3.1.2 Methods for Type 2 Problems

For type 2 problems where generalization performance is an issue, cross-validation should be used to decide when to stop training, as mentioned in Section 2.7. There is

no consensus on how to do this, so a reasonable method was developed for this thesis. This method will be outlined in Section 3.2.

3.2 Simulation Environment

All simulations in this thesis were run using the **Xerion** public domain neural network software [19], with various enhancements written for the purpose of the thesis work. Most enhancements are intended to be made publically available in the future. All C code for the enhancements, along with detailed explanations of their usage, will be included in a separate document at the University of Manitoba's Department of Electrical and Computer Engineering.

The **Xerion** software is a collection of routines to allow for easy construction and monitoring of neural network simulators. The software defines a collection of "objects" (actually C data structures) that implement the important features of any neural network system. These objects are Units (neurons), Links (connections between Units, these have modifiable weights), Groups (of similar Units), ExampleSets (training and test data), and Nets (contain Groups and ExampleSets). The software also contains routines to graphically display Unit outputs and Link weights using Hinton diagrams[2]. A typical display of the simulator is shown in Figure 3.1. The network being simulated had 20 inputs, one layer of 6 hidden units, and 3 output units.

To create a simulator using the software, the user must write a set of routines that implement some neural network algorithm. These routines must include routines for updating the network error, and calculating the derivatives of the links with respect

[2]In a Hinton diagram, each weight, activation, or other entity is represented by a square containing another square. The size of this second square corresponds to the entity's magnitude. A white square means the entity is positive, while a black square means the entity is negative.

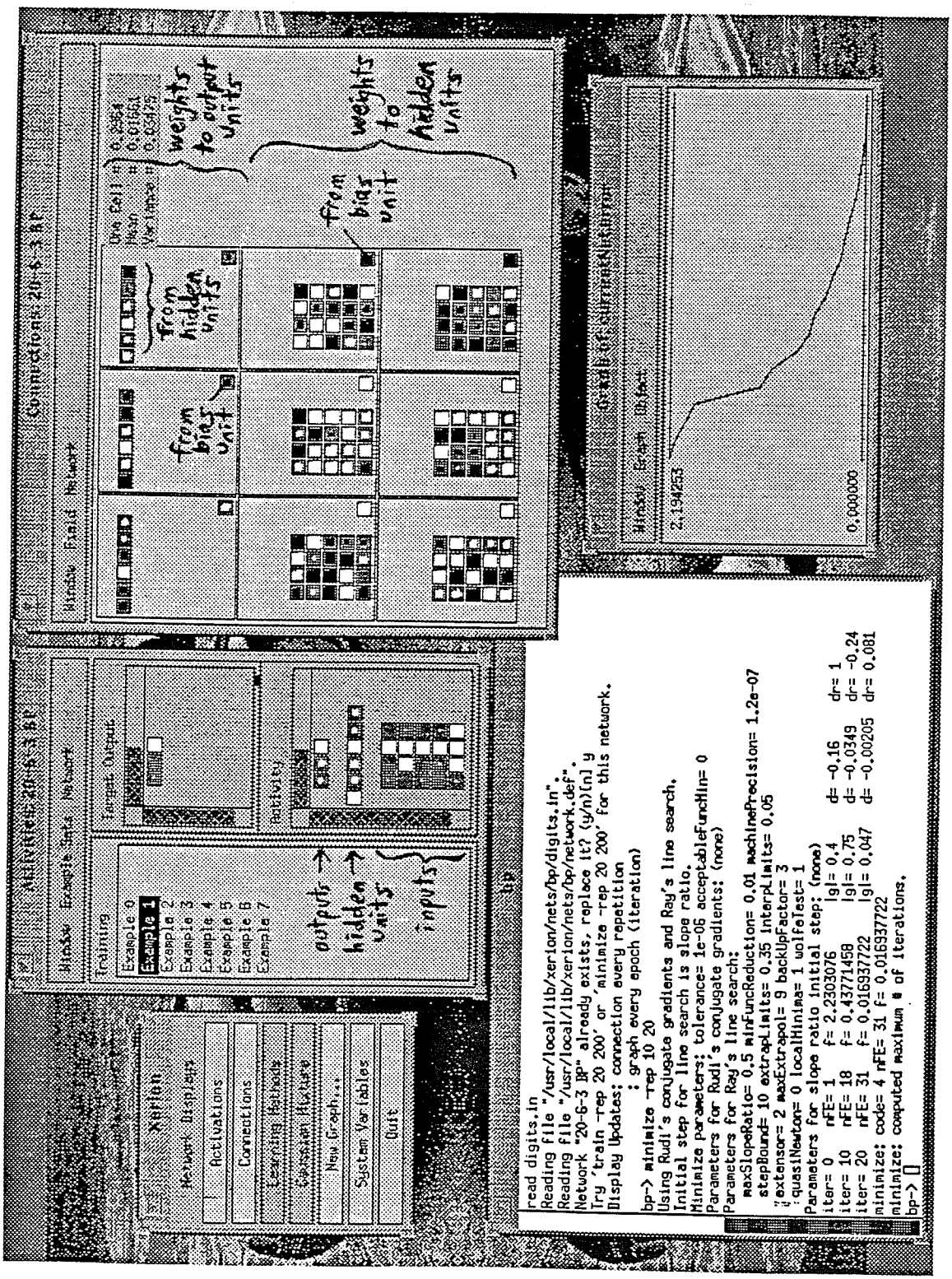

Figure 3.1: Typical Display of Xerion Simulator

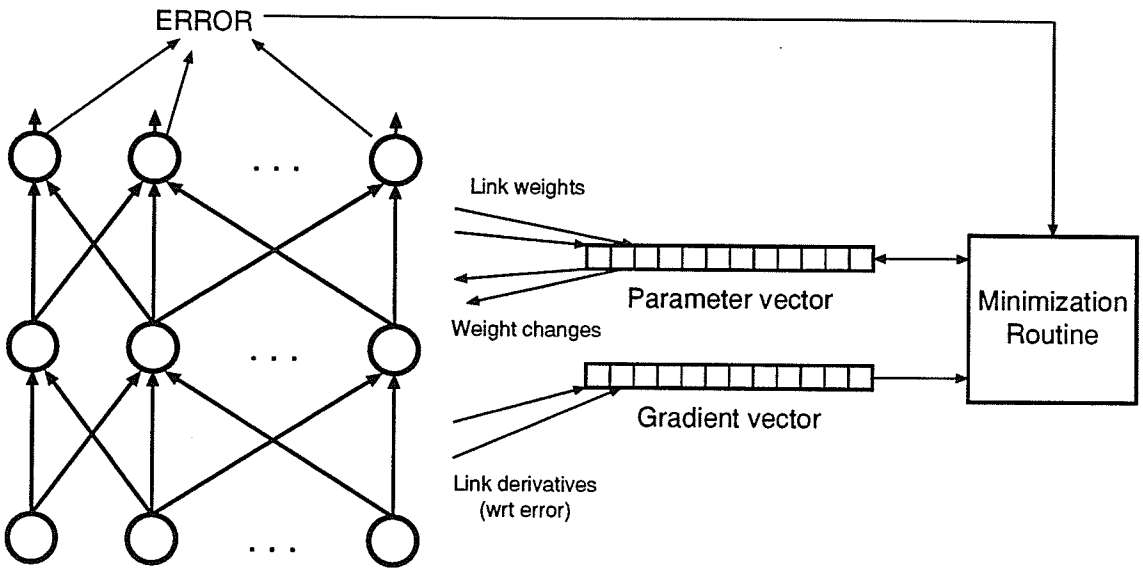

Figure 3.2: Xerion Methodology for Network Training

to the error (ie: computing the gradient). The user may desire the network to store certain values in addition to those already contained in the **Xerion** objects, so the software allows the user to add "extension records" to each object. The user may also add new commands to the simulator, executable from the simulator command line.

One of the main features of the **Xerion** software is a minimization library. **Xerion** converts the link weights into a parameter vector, and the calculated link derivatives into a gradient vector. With these two vectors, and the error function to be minimized, one can use all sorts of optimization techniques, such as the Conjugate Gradient methods described in Chapter 2. **Xerion** does in fact provide a wide selection of minimization routines that one may use, including conjugate gradient routines. The **Xerion** code for conjugate gradient follows the code in Numerical Recipes in C [33]. The minimization methodology is pictured in Figure 3.2.

The **Xerion** software distribution includes several pre-built simulators, one of which implements backpropagation. The code for the backpropagation algorithm

provided a starting point for much of the code in this thesis. For the studies of various minimization algorithms for backpropagation, several additions to the code were made:

1. Routines to implement automated cross-validation. To implement these routines, a **Xerion** facility for calling the minimization routines from within C code was taken advantage of, to allow for separate minimization trials between cross-validation checks. Before the routine to train using cross validation is invoked by the user (through a new command line command), he or she must set all **Xerion** minimization parameters to some desired values. The **Xerion** minimization parameters include such things as *epsilon* (the learning rate in plain backpropagation), *alpha* (the momentum parameter), parameters for line searches, parameters for the delta-bar-delta and quickprop rules, and so on. The user must also select which minimization technique (momentum, quickprop, etc) is to be used. Normally the user need not worry about all minimization parameters at once; some are only for use with a particular minimization method. In addition to the minimization parameters, there are three new parameters that the user must set to control training time for cross-validation:

   - Parameter 1. The number of weight updates between validation checks. By not doing a check after every weight update, computation time is saved. Also, short-term fluctuations in generalization performance may be "filtered out" by looking at performance measures separated in time.
   - Parameter 2. The number of weight updates that the generalization performance can go without improving before the training run is stopped. In

the simulations presented in this thesis, this number allowed for training to be stopped after 4 or 5 consecutive validation checks with no improvement.

- Parameter 3. The maximum number of weight updates allowed.

The cross-validation technique then proceeds as shown in Figure 3.3. Note that generalization performance is measured using the total output error on the test set, and not some other measure like percent correct classification. This is because the network is trying to minimize output error and not percent correct, so the latter will likely fluctuate more often and will not work as easily with the cross-validation technique presented.

2. Generation of "general" sigmoids, and calculations of derivatives for these sigmoids. The general sigmoids are set using three parameters *sigmoidMax, sigmoidMin*, and *sigmoidGain* according to the following equation:

$$\text{Neuron output} = y(x) = \frac{(sigmoidMax - sigmoidMin)}{1.0 + e^{-sigmoidGain*x}} + sigmoidMin \quad (3.2)$$

To generate the logistic function, one would set $sigmoidMax = 1.0$, $sigmoidMin = 0.0$, and $sigmoidGain = 1.0$.

To generate the tanh function, one sets $sigmoidMax = 1.0$, $sigmoidMin = -1.0$, and $sigmoidGain = 2.0$.

The derivative of the general sigmoid is:

$$y'(x) = sigmoidGain * (y - sigmoidMin) * (1.0 - \frac{y - sigmoidMin}{sigmoidMax - sigmoidMin}) \quad (3.3)$$

For the logistic function this derivative works out to $y(1-y)$. For the tanh it works out to $1 - y^2$.

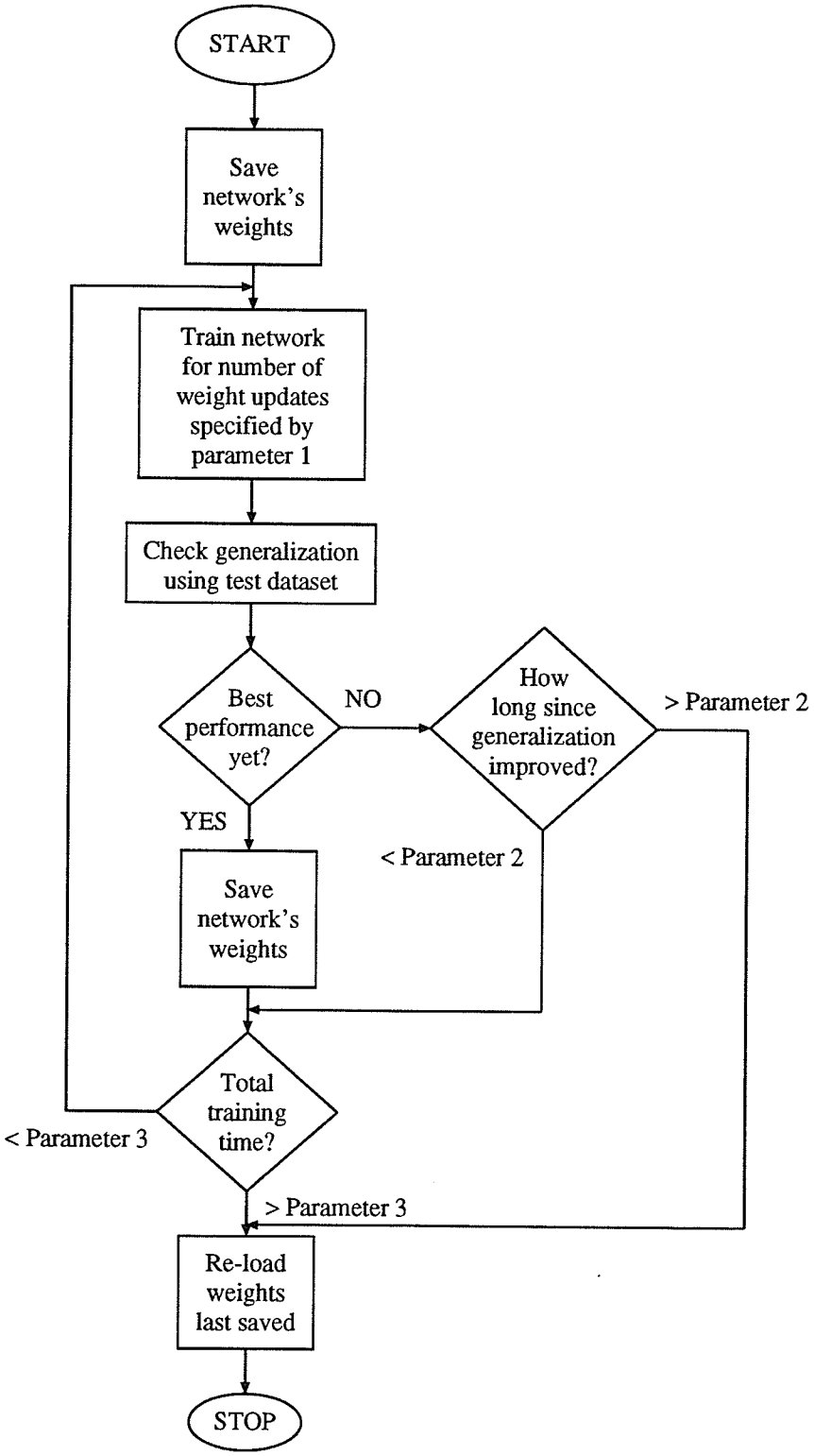

Figure 3.3: Cross-Validation Method

The general sigmoids were added to the code at a time when the only activation function supplied with the **Xerion** software was the logistic. In a newer release of the software a wide variety of activation functions (including the tanh) have been made available.

3. Routines to print classification statistics as percentages correct, incorrect, and unsure. These routines are accessed through a new command line command. It is assumed that one output is used to represent each class, or (for two-class problems only) that only one output neuron is used and it must produce a high output for class 1 and a low output for class 2. The user has three more parameters to set when using the classification statistics routines, each corresponding to a threshold that must be (or must not be) exceeded for a correct classification to occur. These three thresholds were originally described by LeCun et al in [26]:

   (a) The output of the most active neuron (which must correspond to the correct class) must exceed a threshold

   (b) The output of the second most active neuron must be less than a threshold

   (c) The difference in activation levels between the two most active neurons must exceed a threshold

   For an *incorrect* classification to occur, all three criteria must be met except that the most active neuron does not correspond to the correct class. If neither a correct or incorrect classification occurs, the network is said to be "unsure" of the class of the input pattern. The user may view only a summary of classification statistics (percents correct, incorrect, and unsure for each class), or

Table 3.1: Output classification for single output neuron

	Target value position within neuron range	
Output activation	lower half	upper half
< (MIN+MAX-threshold)	CORRECT	INCORRECT
> threshold	INCORRECT	CORRECT
otherwise	UNSURE	UNSURE

may view the classification on a pattern-by-pattern basis. In the latter case, activations of the most active neuron, second most active neuron, and neuron corresponding to the correct class are also printed for each pattern. This allows for more detailed analyses such as what threshold 3 above must be increased to in order to get the percentage misclassified down to, say 1% (while rejecting more patterns).

For the single output neuron case, only the first threshold comes into play. Correct, incorrect, and unsure classifications are determined according to Table 3.1 (MIN=sigmoidMin, MAX=sigmoidMax):

4. Proper support for updating weights after each pattern presentation. This includes new commands to create pseudo-random sequences in which to present the patterns (pseudo-random because each pattern is guaranteed to be presented at least once each training epoch, and the order of pattern presentation is the same each epoch). The random sequence capability is very important because it allows for proper pattern-by-pattern learning on datasets that contain all examples of one class, then all of another, and so on. If such a dataset were read sequentially, the network would tend to set itself up for classifying the first class only, possibly letting the weights get very large, thereby saturating the sigmoids and making further learning very difficult.

In addition to these enhancements, one important change was made. The original **Xerion** code used derivative of output error wrt output neuron activation $\partial E/\partial V_i = 2(V_i - V_i^d)$. The factor of 2 was dropped to conform with common practice.

For the study of the cascade correlation algorithm, the code for the complete algorithm was written for the **Xerion** simulator as none was yet available. The cascade correlation algorithm requires two phases of training. In the first phase, the weights of the output units' incoming connections are updated, and in the second phase the weights of the candidate units' incoming connections are updated, using a different method of calculating weight error derivatives. These phases of learning alternate as the network is being trained. To keep cascor training as simple as possible for the user, it was decided to create a new command that when issued will automatically run the network through the various phases of training and automatically add hidden units. The different procedures used for updating candidate and output unit weights were accessed by assigning the network's fields for error updating and link derivative updating procedures to point to the appropriate procedures.

The performance measure for candidate unit training (the covariance calculation of equation 2.21) must theoretically be maximized, but **Xerion** expects to be minimizing a function. To fool **Xerion** into thinking it was minimizing a function during candidate unit training, the performance measure implemented was the *negative* of the covariance (plus, of course, additional network costs such as weight size).

Candidate unit training requires two sub-phases of learning: The first is to calculate the mean errors and mean candidate unit activations, and the second is to calculate the weight error derivatives based on these means. Two passes through the training data are therefore required before weight updates can be made. The original

Cascade Correlation code from Carnegie Mellon University [27] actually calculated weight error derivatives at the same time the means were being computed, with no guarantee that this method of saving computation would work. This method was not implemented; the "pure" algorithm in [10] was implemented instead.

Since the **Xerion** minimization routines only require weight error derivative calculations, and the Cascade Correlation routines provide these, it is possible to use any minimization method in the **Xerion** library for training. Because the algorithm consists of two distinct phases of training, it was decided to provide for the user to use different minimization methods for output and candidate unit training. It is therefore possible to use, say, the delta-bar-delta rule for output unit training and a conjugate gradient method for candidate unit training.

The enhancements made to the code for backprop were linked with the cascor code. Automatic cross-validation is done during output unit training in the same way as for backprop (by checking the network's output error on the test set), and is done during candidate unit training by checking the candidate unit outputs' covariances with the output error on the test set. Nine parameters are used to control training time for cross-validation — three for each of the initial output unit (no hidden units), subsequent output unit, and candidate unit training phases (the three parameters used per phase are described on page 37). The initial output unit training phase is treated separately because it generally takes longer than subsequent output unit training phases. Not all parameters have to be set — if no parameters are set for the candidate and/or initial output unit training phases, the parameters will automatically be set to the ones used for second and subsequent output unit training phases.

3.3 Results

3.3.1 Cereal Grain Classification

The reader may question how much better neural networks are at pattern classification problems than more traditional classification techniques. This section will show that in certain cases such as the classification problem presented in this section, neural networks are superior to certain traditional methods.

The classification problem is that of distinguishing between various cereal grain kernels (barley, oats, rye, and amber durum, hard red spring, and soft white spring wheats) using data obtained through optical measurements [59]:

1. Light reflectance data: red, green, and blue light
2. Size data: length, width, contour length (perimeter), three different aspect ratios, and area.

Previous work [59, 65] that used only light reflectance data or size data yielded much poorer results than those obtained using all data that will be presented in this section.

Distinguishing between grain kernels of different types is part of an ongoing larger project at the University of Manitoba. Its objective is the separation of sound hard red spring wheat kernels from other types of grain kernels (the problem studied here) and also unsound wheat kernels.

The results obtained for grain classification using neural networks will be compared to results obtained by another member of the department, Mike Neuman, using a Gaussian maximum likelihood classifier (operation of the latter described in Appendix A). The grain classifier input data consisted of ten data points, each corresponding to one of the optical measurements listed above. Data from a total of 727 grain kernels

were available, one-third (242) of which were used for training the network and the other two-thirds (485) of which were used for testing generalization performance. All data for the neural network were normalized to lie within [0,1]. This ensured that the data were on the same order of magnitude as the output of the virtual bias neuron (i.e. 1.0), so that the neuron biases could be learned properly.

The biggest problem faced when trying to solve the grain classification problem (or any problem, for that matter) with a neural network is that of selecting an appropriate number of hidden units. Before the cascade correlation algorithm was properly implemented, a series of training runs was performed with multi-layered perceptrons each with one hidden layer of neurons and six output neurons, one for each type of grain. Only one hidden layer was used because it was believed, based on plots of the data, that data from each class (grain) formed a single cluster in input space [59]. As was shown in section 2.1, only one hidden layer of neurons is necessary for such a problem. The number of hidden neurons was varied from zero to ten. Both logistic and tanh neurons were tried in an attempt to see whether use of logistic neurons really does hamper learning, mentioned as a possibility in Section 2.4. For each number of hidden units, ten training runs were performed, each with a different set of starting weights (in the range (-0.5, 0.5)), because the choice of initial weights can affect the final outcome [64]. The total number of training runs was therefore 220.

The training runs were done using the enhanced **Xerion** simulator, using back-propagation with only the default minimization algorithm supplied, Rudi's conjugate gradient with Ray's line search. The main purpose of the neural network grain classification simulations was performance comparison between neural networks and a classical technique, and not comparison of neural network minimization algorithms,

which is why only one minimization algorithm was used. The parameters for controlling the line search were the **Xerion** defaults, given in Appendix B.

The three parameters controlling training time for cross validation were set to 50 weight updates between cross validation checks (note that the output error and gradients might have to be measured more than 50 times to get 50 weight updates, because of the line searches), 200 weight updates before an improvement in generalization had to be noticed, and 500 maximum weight updates allowed. Cross-validation parameters for all thesis work were set so that the weights yielding best generalization were not found after too few or too many validation checks. Target values for the tanh networks were -0.7 or 0.7, while for the logistic networks they were 0.2 or 0.8. A classification was considered correct when the output neuron corresponding to the correct class had activation greater than the others; otherwise it was incorrect.

Generally the accuracy on the training data was about 92% for 5 or more hidden units, with less accuracy for less hidden units. The logistic networks had more trouble with low numbers of hidden units. Accuracy on the training data for the Gaussian classifier was 94.2%. Results showing average percent correct, best percent correct, and worst percent correct on the test set are plotted in Figure 3.4 for the logistic and tanh type networks with number of hidden units ranging from 0 to 10. Using more than 5 hidden units did not improve the generalization performance by much; in fact the best average generalization performance (for both types of neurons) was reached at 6 hidden units. This implies that many more hidden units than this is either unnecessary for this problem, or more training data would be needed to achieve better accuracy with a larger network. Nevertheless, the small gap between accuracies on training and test data indicates that one-third of the data was adequate

for training the networks. Less would probably have resulted in poorer generalization performance, especially for the larger networks. The 6-hidden unit networks trained on 242 patterns were being trained on roughly 2 patterns per weight.

The generalization accuracy for the logistic networks was worse than the tanh networks for low numbers of hidden units, but about the same for high numbers. This and the fact that the logistic networks consistently took longer to train shows that the logistic networks are indeed harder to train. Average training times, in **function evaluations** (one function evaluation = one output error and weight error derivative $\partial E/\partial w_{ij}$ (gradient) calculation, possibly accumulated over all patterns (if using batch updating)) and **iterations** (multiplications of weight error derivatives by learning rate etc. followed by actual weight updates) are plotted in Figure 3.5. The training time decreases either when a very low number of hidden units is used and the problem is too hard (the network "gives up" on it quickly), or when a very high number of hidden units is used (possibly due to premature saturation of the output units because of large net inputs resulting from large fan-ins).

It turns out that for these networks, learning often stopped abnormally, either when the line search algorithm failed, or when the simulator detected an inconsistency in the calculations due to too low a precision (the simulator was compiled 32 bits floating point on a Sun SPARC workstation). When this happened, however, the validation error was changing so slowly that this had little effect on the results. It was regarded as a time saver because time would have been wasted waiting for very small improvements in performance to occur. The **Xerion** manual states that training will stop due to low precision only when a minimum of the error function is very near. The tanh networks seem to have reached these minima sooner. The logistic networks

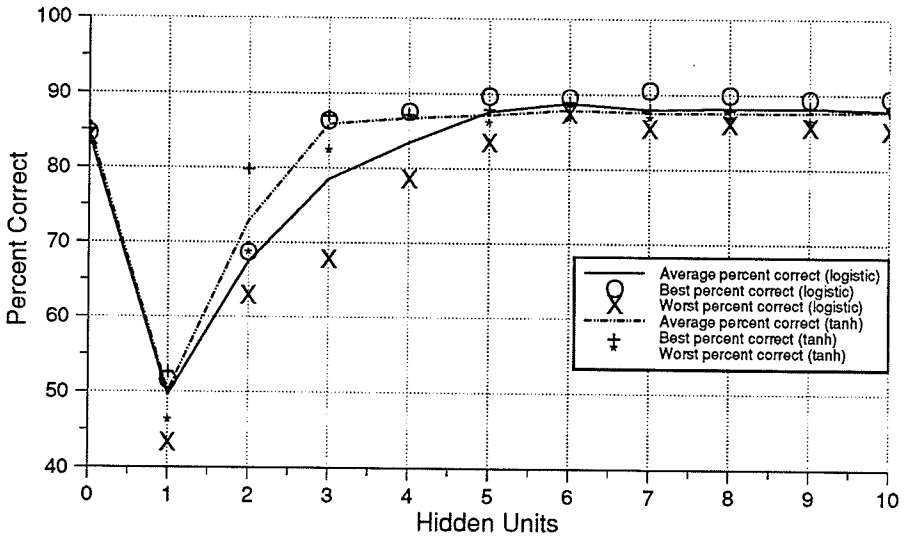

Figure 3.4: MLP Generalization Performance on Grains Problem

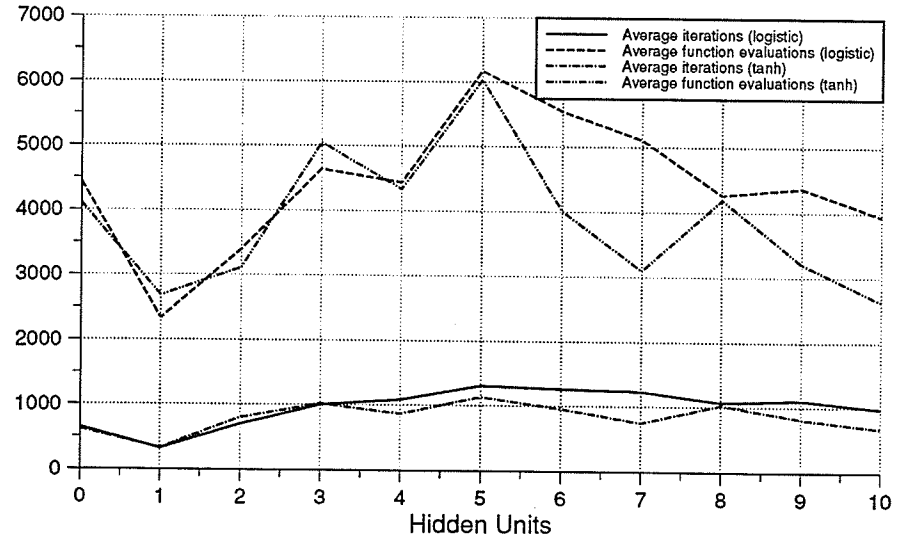

Figure 3.5: MLP Training Time for Grains Problem

Table 3.2: Cross Validation Parameters for Cascade Correlation Grain Classifier

	Phase of training		
Iterations...	Initial (no hidden units)	Candidate	Output
between validation checks	50	10	1
for generalization improvement	200	40	5
Maximum iterations allowed	1000	200	50

with few hidden units seem to have stalled in poorer local minima.

The networks with at least 6 hidden units consistently generalized better than the Gaussian maximum likelihood classifier. Average total percent correct on the test data for the Gaussian classifier was 85.2%, as compared to 87.9% for the tanh networks with 6 hidden units and 88.8% for the logistic networks with 6 hidden units. It appears that the assumption of Gaussian-distributed data was not as good for generalization as the neural networks' lack of assumptions about the data distribution.

In addition to the backpropagation neural nets and Gaussian classifiers, attempts were made to classify the grains using cascade correlation networks. These networks were allowed to add up to ten hidden units. All training was done using Rudi's conjugate gradient with Ray's line search. Ten runs were done for cascade correlation networks with either logistic or tanh neurons, for a total of 20 training runs. Parameters for controlling cross validation are given in Table 3.2.

A rather surprising result was that the networks with only one tanh hidden unit yielded very good classification accuracy. The networks with logistic neurons yielded consistently poorer accuracy. A plot of average percent correct on the test data is shown in Figure 3.6. Training past the initial phase (no hidden units) proceeded very swiftly—tens of iterations as opposed to hundreds. Once again training often stopped close to minima, where the simulator reported loss of accuracy. A plot of average training time is shown in Figure 3.7 for the tanh networks only; the logistic

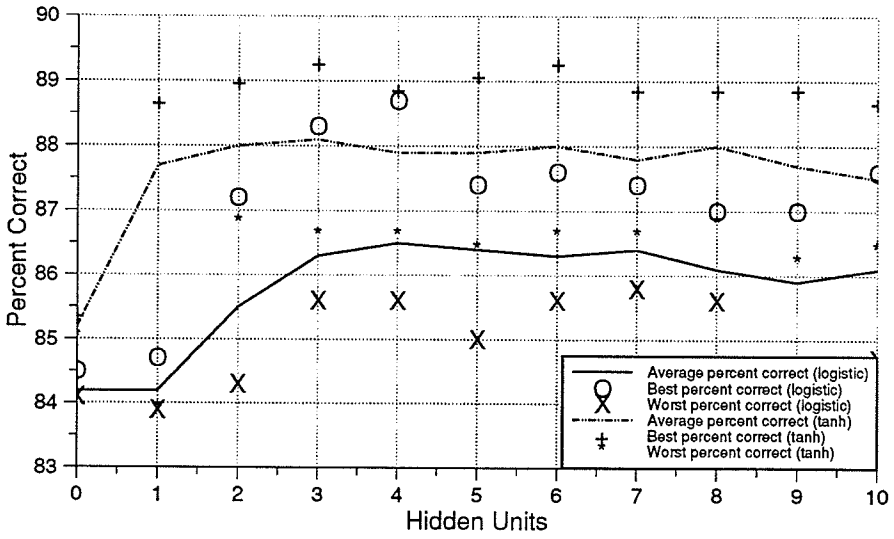

Figure 3.6: Cascor Generalization Performance on Grains Problem

networks took slightly longer to train. The plot shows the average *cumulative* number of iterations and function evaluations to train up to each added hidden unit. Iterations and function evaluations for the candidate and output units are plotted separately, because of the differing amount of computation involved in training these different units.

It is clear that using the cascade correlation approach leads to compact networks that achieve the same accuracy as MLPs with larger numbers of hidden units. They may still have more weights than MLPs with similar accuracy, however, because of their "shortcut" connections, direct connections between input and output. These connections make the structure of the network less modular (and therefore more difficult to implement in hardware), but nevertheless add processing power to a network with a low number of neurons.

For a fairer comparison between a cascade correlation and backpropagation-trained

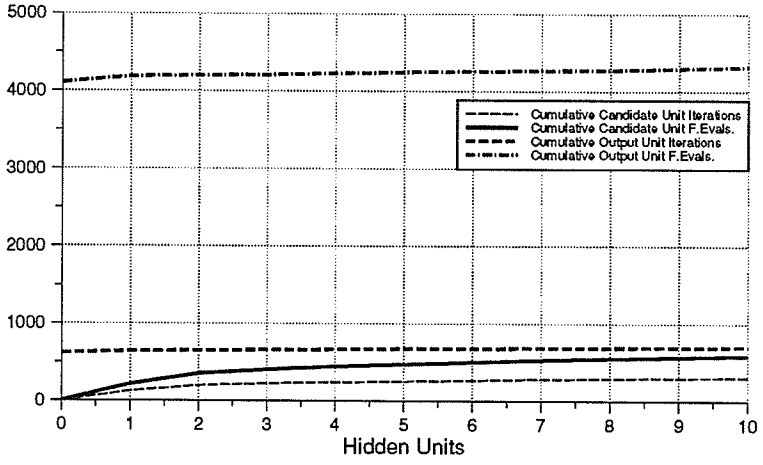

Figure 3.7: Cascor Training Time for Grains Problem (tanh units)

network, it was decided to run another set of training runs with backpropagation-trained networks with shortcut connections. Note that cascade correlation still has the advantage of connections between hidden units. A total of 220 more simulation runs were performed in the same way as for the MLPs, except that shortcut connections were used. Plots of generalization performance are shown in Figure 3.8. On average the tanh networks did not perform as well as their cascade correlation counterparts until about 5 hidden units (the logistic cascade correlation networks always performed worse). Once again peak accuracy was reached at 6 hidden units. The logistic networks performed only slightly better than the tanh ones. On average around 1000 iterations and 5000 function evaluations were needed to train the networks. The total number of iterations and function evaluations to train up a similar sized cascade correlation network from zero hidden units was about the same, and the total computation may be lower because of the fewer weights being trained at any one time. Also, one will automatically see the performance of intermediate-sized networks (use-

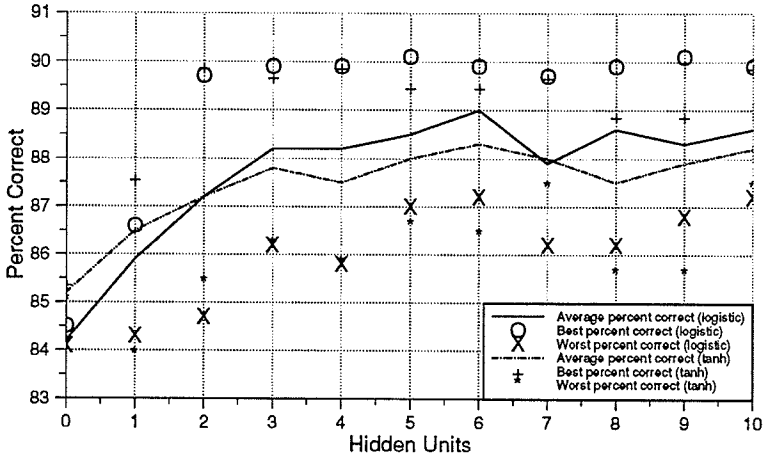

Figure 3.8: MLP with Shortcuts Generalization Performance on Grains Problem

ful because the network designer can subsequently train up to the intermediate-sized network beyond which no performance increase was found), whereas looking at performance of each intermediate-sized MLP would require another 1000 iterations and 5000 iterations per size. Cascade correlation therefore saves time. It is also the best type of classifier for very low numbers of hidden units (< 3). The algorithm seems to have simplified the classification problem for these small networks by solving the moving target problem. However, it is really number of weights and not number of hidden units that should be compared. The tanh MLPs with 3 hidden units and no shortcut connections, which performed comparably to the tanh cascor nets with 3 hidden units, had fewer weights than even the cascade correlation networks with no hidden units.

A summary of the best average generalization results obtained for all types of classifiers studied is shown in Table 3.3, along with the hidden units and weights required to achieve that accuracy (in parentheses in the column headings). The

Table 3.3: Summary of Best Average Grain Classification Generalization Results

Grain	MLP		MLP shortcut		Cascor		Gaussian
	L(6,108)	T(6,108)	L(6,168)	T(6,168)	L(4,140)	T(3,120)	
Barley	93.4	91.6	93.3	93.4	91.0	92.4	91.8
Oats	90.8	90.4	89.7	89.6	88.2	87.5	94.4
Rye	97.4	97.9	98.5	98.2	96.7	97.5	100.0
A.D.	84.1	83.0	85.3	82.8	79.4	82.3	79.8
H.R.S.	95.3	95.4	96.3	96.2	95.7	97.7	91.8
S.W.S.	72.2	68.7	70.8	69.7	68.2	70.6	54.4
Overall	88.8	87.9	89.0	88.3	86.5	88.1	85.2

L = Logistic, T = Tanh, A.D.= Amber Durum, H.R.S.= Hard Red Spring, S.W.S.= Soft White Spring

neural networks clearly achieved better generalization performance overall than the Gaussian classifier, especially when it came to classifying hard red and soft white spring wheats. The results do not indicate, however, that the Gaussian classifier was a poor choice of classifier—it did, in fact, do a better job at classifying oats and rye.

3.3.2 Handwritten Character Recognition

This section will present results obtained on a large classification problem: that of classifying handwritten numerals. For this study, a database of handwritten digits was obtained from the University of Windsor containing roughly 4000 numerals in various handwriting styles. The database originated at the French company CGA-Alcatel, where participants in the study were asked to enter a random sequence of disconnected digits in a specially designed form [34]. As a result of the database's European origin, many of the "1" 's seem rather odd-looking from a North American perspective, appearing as inverted "V" 's. Some can be seen in the small sample of the database shown in Figure 3.9.

When using a neural network to classify an image such as a handwritten digit, one of two approaches can be used. The first is to do extensive pre-processing to

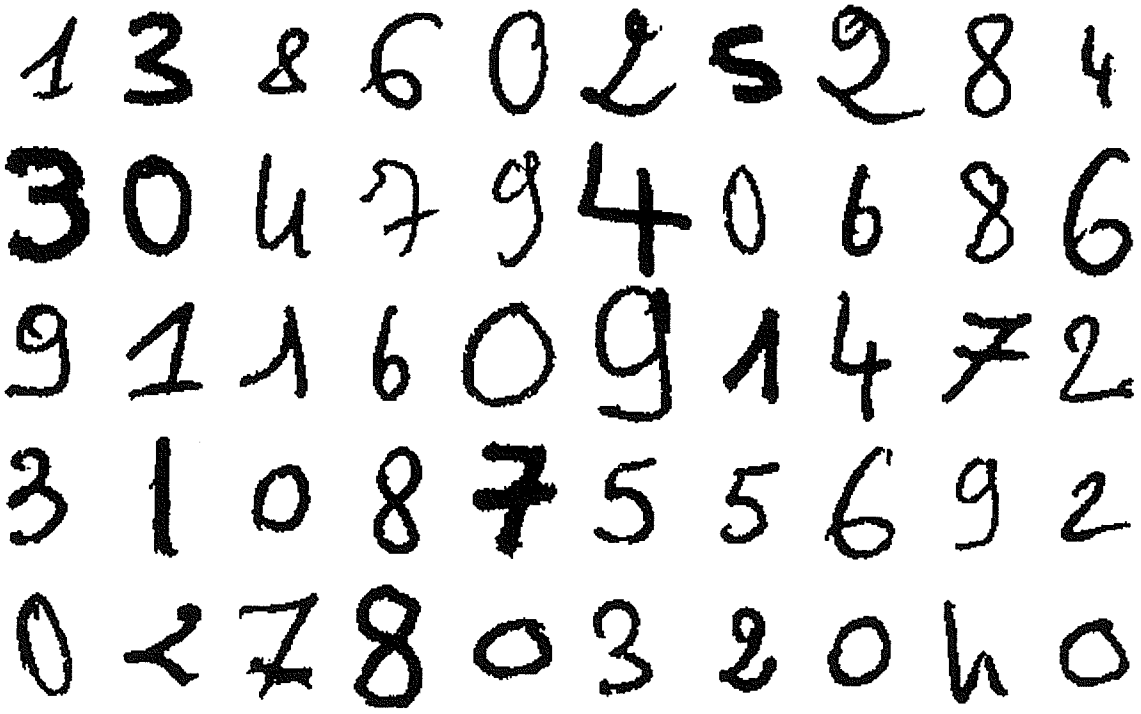

Figure 3.9: Sample of Handwritten Digit Data

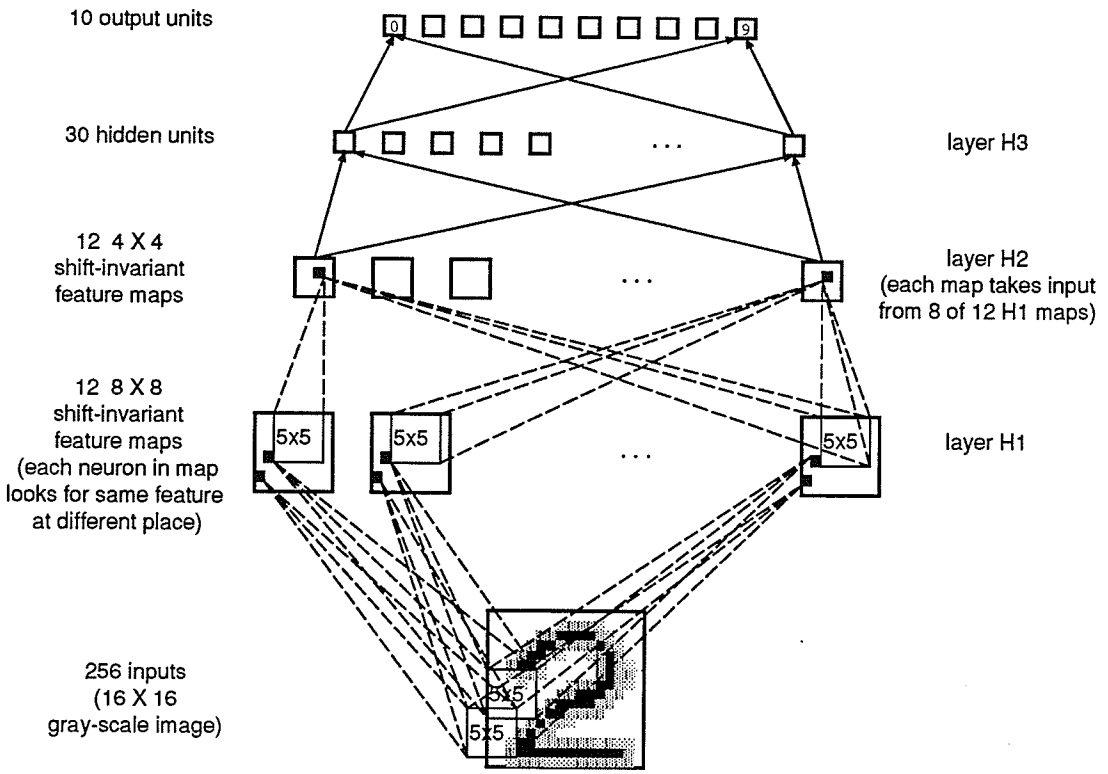

Figure 3.10: Network Architecture for Handwritten Character Recognition

extract features such as chain code histograms[3] and then train a very simple network on the extracted features, such as was done at the University of Windsor [35]. The second method is to train a specialized network on data that has undergone minimal preprocessing, and force the network to extract features of the input on its own. A specialized network for handwritten digit recognition was proposed by a group at AT&T Bell Labs [25]. It is a hierarchical network, with neurons in the "lowest" layer looking only at small parts of the image. Neurons in "higher" layers then combine the signals from the lower layers, until finally a decision can be reached as to the correct classification. The network's architecture is shown in Figure 3.10.

The network takes as input a 16-by-16 grayscale representation of the input image.

[3]A chain code histogram is constructed by first extracting the contour of the image, and then determining the frequencies of various angles of orientation of the contour.

All inputs range from -1 to +1, and all neurons have tanh transfer functions.

The hidden layer directly above the inputs is organized into 12 8x8 "feature maps". Each neuron in a feature map receives input from a 5x5 section of the input image. Neighboring neurons in a feature map take input from a 5x5 section displaced by 2 inputs. Connections extending beyond the limits of the input array are assumed to be attached to "virtual" neurons with some fixed output. In the original AT&T network, the virtual neurons had output -1. All neurons within a feature map have their input weights constrained to be the same as the input weights of all other neurons in the feature map. Bias values are allowed to vary, however. The purpose of the weight constraining is to allow the network to look for the same input feature in different parts of the input. The varying bias terms allow the network to assign a feature different degrees of importance depending on its location within the image.

The neurons in the second layer are organized into 12 4x4 feature maps. Each neuron takes as input 5x5 sections of 8 out of 12 feature maps in the first layer. All neurons within a second layer feature map take input from the same 8 first hidden layer feature maps. Weight constraining is done as for the first hidden layer neurons. The connection scheme for first to second hidden layer feature maps is (from [36]):

H2 Feature Map	H1 Feature Map 111 123456789012
1,2,3	XXXX XXXX
4,5,6	XXXX XXXX
7,8,9	XXXXXXXX
10,11,12	XX XXXXXX

(X = Connected)

A third hidden layer containing 30 neurons is fully connected to all neurons in the

second hidden layer, and the third hidden layer neurons are in turn fully connected to all output neurons.

The network was replicated using the **Xerion** simulator, and used on a processed version of the University of Windsor dataset. The original dataset contained characters represented by up to 64x80 binary pixels. These were converted into 16x16 grayscale representations using linear transformations that preserved the original characters' aspect ratios. Basically the transformations were equivalent to dividing the regions containing the characters into 16x16 zones, and coding the data for the neural network according to how much "black" was within each zone (-1 = no black, +1 = all black).

The dataset was divided up into two subsets for training and testing. Each subset contained roughly equal numbers of examples of each digit (ie: the network would be trained on 50% of the digits and tested on the other 50%). Each example added to a subset was selected at random from the original total database of ~4000 digits.

Xerion supports weight constraining, and treats each group of constrained weights as one parameter of minimization. Gradients for each weight in the group are calculated normally using the backpropagation algorithm, and these are summed to get the contribution to the gradient for the single minimization parameter. This parameter, along with its contribution to the gradient, is used by the **Xerion** minimization routines (momentum, quickprop, etc). The resulting calculated weight change is then made to all weights in the group. This method of using one minimization parameter in place of many saves computation. All weights were initialized in the range $\pm 2.4/(\text{neuron fan-in})$, as in [25] so that the net inputs to the neurons could stay within an acceptable range to avoid neuron saturation (and little error passed back).

Table 3.4: AT&T Net: Simulation Parameters

	Conjugate Gradient	Plain BP
Learning rate	N/A	0.02
Iterations...		
between validation checks	5	9720
for generalization improvement	20	38880
Maximum iterations allowed	500	972000

Preliminary simulations initializing the weights in (-0.5, 0.5) did not work nearly as well; final total squared output errors observed on the test data were roughly double those that will be shown in Figure 3.11.

There were two differences in training procedure from that used in [25]. First, the output of the virtual neuron was kept at zero instead of -1. In this way, portions of neuron receptive fields falling beyond feature map boundaries do not influence learning. This change was found to roughly double the learning speed. On preliminary simulations conducted using initial weights in (-0.5, 0.5) and conjugate gradient training, the total squared output error on the training data took 102 and 58 weight updates to fall below 1000.0 using the -1 and 0 virtual neurons respectively. Final generalization performances were similar. Second, the minimization algorithms used to train the weights were different. The AT&T group used the pseudo-Newton method with on-line updating, while here either Rudi's Conjugate Gradient with Ray's Line Search with batch updating, or plain backpropagation with on-line updating was used. Simulation parameters are shown in Table 3.4. Plain BP will be updating weights more frequently, which is why its entries in the table appear so high (9720 = 5 x number of training patterns). Two training runs were done for each of the minimization methods used, each run requiring several days of CPU time on a SPARC 2 workstation.

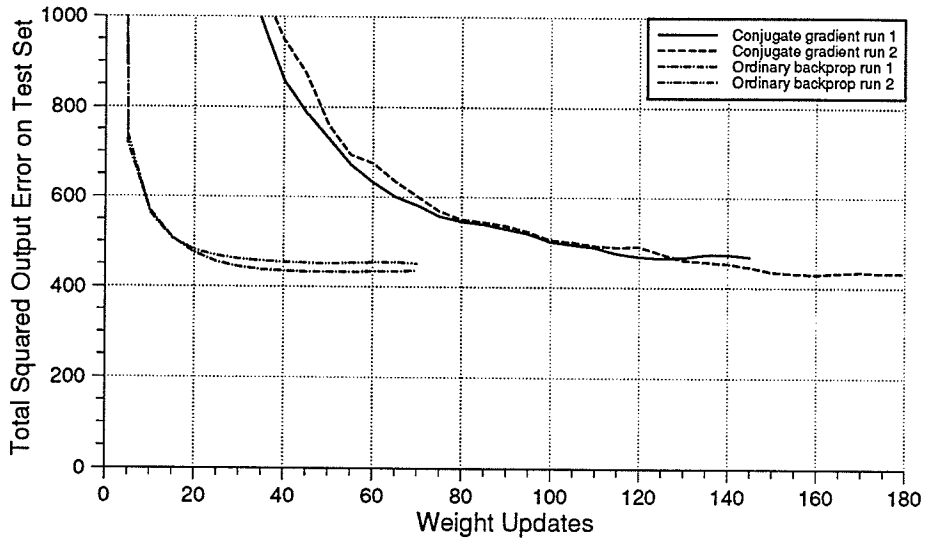

Figure 3.11: Learning Curves for AT&T Net

The training data was consistently learned more or less perfectly. Learning curves showing error on the test set are shown in Figure 3.11. The curves show error versus weight updates, which is being kind to conjugate gradient as there is no indication of extra line search function evaluations not followed by weight updates. The number of function evaluations was actually about three times the number of weight updates. The curves show that the error on the test set starts to level off after a certain period of time. Training always stopped due to this lack of improvement in generalization error, and never due to lack of precision in the simulator. The curves also show that plain backpropagation with on-line updating found a solution similar in quality to that found by conjugate gradient, but required much less computation. For this problem, the redundancy in the training data obviously made it wasteful to accumulate weight error derivatives over all patterns.

In an application such as this, one can set bounds on how many misclassified

patterns are acceptable out of the patterns that are not rejected as unclassifiable (uncertain). By adjusting the classification criteria outlined on page 40, one can determine how many patterns must be rejected as uncertain to get that amount of error. In this study, as in [25], the classification criterion adjusted was the minimum acceptable difference between the two most active outputs.

For 1% error on the classifiable patterns, the percentage of patterns rejected was 10.6 and 10.9 for the conjugate gradient-trained networks, and 11.7 and 11.8 for the networks trained on-line. These figures compare favorably with the result of 12.0% reported by the group at Bell Labs using three times as much data for training, and roughly the same amount of data for testing [25]. In a more recent publication, they used a modified version if their original network and needed 9% rejections for 1% error [26]. The group at the University of Windsor required only about 3% rejections for 1% error using simple networks trained using chain code histograms, and twice as much data for both training and testing [35].

It would appear that humans are still better at feature extraction than supervised ANNs. Nevertheless, ANNs are still able to extract meaningful features. Figure 3.12 shows Hinton diagrams of the learned input weights for neurons in each of the 12 first layer feature maps. These weights are the "features" that the network, in this case the first network trained using ordinary backprop, has learned. These features, when convolved with the input image, can be used as edge finders etc. For example, the feature second from the right in the bottom row of Figure 3.12 can be used to detect a horizontal edge. A neuron with these weights will produce a strong positive output when a horizontal edge with handwriting on bottom, whitespace on top is observed, due to the positive correlation between the input image and the weights.

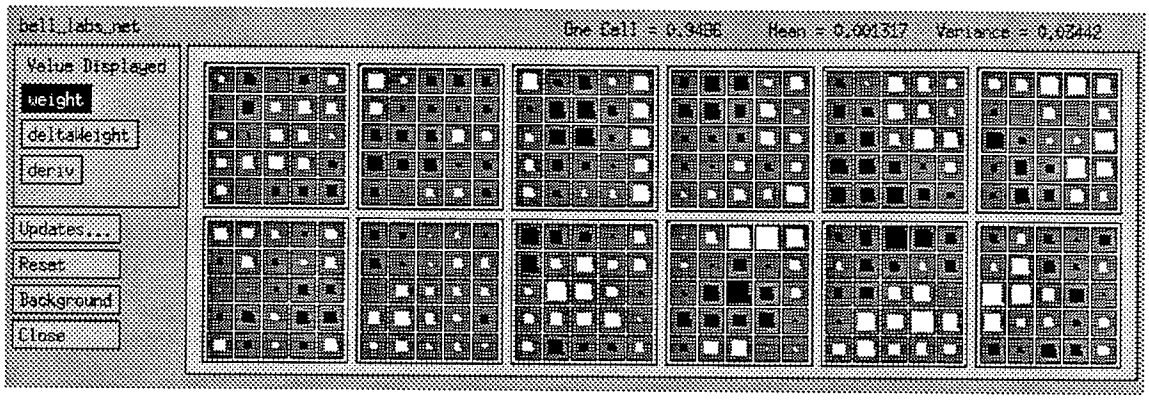

Figure 3.12: Learned Features for Handwritten Character Recognition

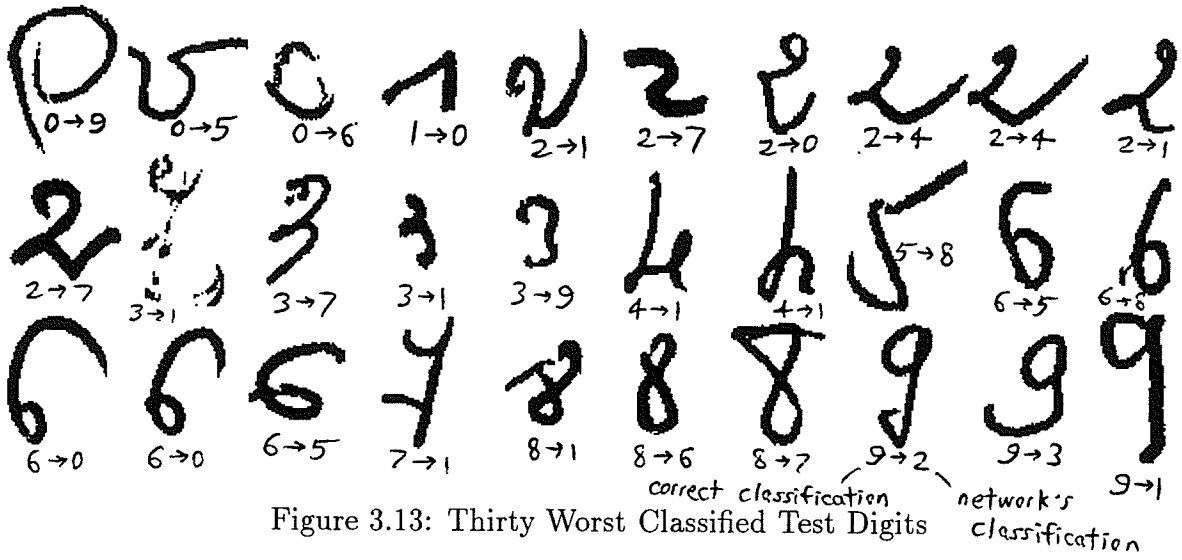

Figure 3.13: Thirty Worst Classified Test Digits

Figure 3.13 shows the same network's 30 "worst" classified digits in the test set—those which the network misclassified with the highest degree of confidence (where the difference between the (incorrect) most active output neuron and second most active neuron was greatest). Humans would have trouble classifying some of these, but not all.

Some additional simulations were done to see how well the cascade correlation algorithm performed on this problem. This algorithm produces simple networks compared to the one just presented, but other authors have successfully built simple

Table 3.5: Cross Validation Parameters for Cascade Correlation OCR Net

	Phase of training		
	Initial (no hidden units)	Candidate	Output
Learning rate	0.0005	0.006	0.0005
Iterations...			
between validation checks	1944	5	1944
for generalization improvement	9720	20	9720
Maximum iterations allowed	38880	50	38880

networks for this problem. In [37] an interesting training technique (not backpropagation) was used to construct a network with only one layer of 45 hidden units. The hidden neurons of this network were each explictly trained to separate inputs of two classes from one another. The output neurons then took ANDs of certain hidden neuron outputs to come up with a classification decision. Only 3% of the patterns in the test dataset had to be rejected to achieve 1% overall misclassification (adjustment of pattern rejection criteria was done differently than Bell Labs' method). In contrast, a 40 hidden neuron MLP trained using backprop has been reported as rejecting 19.4% of the test patterns for 1% error on the remaining patterns [25]. It is obvious that the training technique, and not the network architecture, has a great effect on the results for this problem.

The best results for cascade correlation were obtained using a network that used ordinary training (gradient descent with fixed learning rate) with on-line updating for learning the output weights, and quickprop for learning the candidate weights. Once again all weights were initialized to within $\pm 2.4/(\text{neuron fan-in})$. Some simulation parameters are shown in Table 3.5 ("learning rate" for quickprop is the ϵ term in equation 2.20). Training the network up to 30 hidden units required approximately 1 week of CPU time on a SPARC 2 workstation.

It was noticed that after about 15 hidden units were added, performance did not

improve significantly. With 30 hidden units, the cascor network required 41.6% rejects for 1% error on remaining. A MLP with one layer of 30 hidden units (trained using conjugate gradient) performed better: 35.5% rejects for 1% error on remaining. It appears that cascade correlation is not an appropriate learning algorithm for large problems such as this that require careful solution, because the network may "commit" itself to a poor result early in learning. Backpropagation networks may be able to consider more possible solutions during learning. However, if backpropagation is to be employed for a problem like this, character recognition with limited preprocessing, then a hierarchical type network is appropriate.

3.3.3 Parity and Sonar Signal Classification

Benchmark simulations of many of the backpropagation algorithm enhancements included with the **Xerion** software were done on two problems — one "type 1" problem, and one "type 2" problem. The type 1 problem was studied to observe the differences in learning rate that different minimization networks can achieve. The type 2 problem was studied to observe whether different minimization methods can lead to better generalization in addition to being faster.

The type 1 problem was a parity problem, well known to researchers doing performance evaluation of neural networks. Here the network is presented with binary inputs, 0's and 1's, and the single output neuron must output a high value if the input pattern has odd parity (an odd number of 1's), and a low value if the inputs have even parity. The parity problem is one of the most difficult problems for an artificial neural network to solve, because the network must look at *all* the input signals (and weight each appropriately) in order to determine whether the pattern has even or odd parity. The parity problem can be scaled to have as many inputs as is desired. Four

inputs were used in this study. It is known that a network with four hidden units can solve this problem [1], but 8 were used to ensure more successful trials.

Due to time limitations, all learning parameters for all learning algorithms were not optimized. Instead, all parameters with the exception of the steepest descent parameter (ϵ in equation 2.2), were set to the default values given in Appendix B, because these parameters are generally thought to be quite robust. Ray's line search was used for the conjugate gradient methods, and also for steepest descent with line search (listed as "steepest descent LS" in the tables). Delta-bar-delta in **Xerion** uses equation 2.15 instead of 2.14.

An optimal value of ϵ was found through experimentation. Several trials were done using ordinary backprop with each of many different learning rates, and the average time required for the squared output error on all patterns to drop below 0.16 was noted. It was decided to use $\epsilon = 0.05$ (actually $\epsilon = 0.10$ worked slightly better but learning rates much above 0.10 resulted in disaster, so 0.05 was used for safety's sake). The parameter ϵ was set to 0.05 for all minimization algorithms requiring it, namely ordinary backprop, momentum, delta-bar-delta, and quickprop. This value for ϵ was obtained using tanh neurons. The optimal learning rate for logistic neurons was found to be at least 1.0. The fact that networks with tanh neurons converged rapidly with a lower learning rate may not seem like a significant result here, but it becomes significant when learning rate is actually a length of time to charge a capacitor, as will be the case in a hardware circuit analysed in Section 4.2. As a result of their good performance with lower learning rates, a "hardware compatibility" issue for certain circuits, it was decided to use neurons with tanh functions for all further simulations.

The results given in Table 3.6 are the averages over 30 training runs of the num-

Table 3.6: Average Learning Speeds: 4-bit parity network (8 hidden units)

Minimization method	F.Evals.	Iterations	Failures
Ordinary backprop	836	836	0
Momentum	204	204	2
Delta-bar-delta	240	240	0
Quickprop	417	417	0
Regular CG	666	76	5
Rudi's CG	410	64	7
Steepest descent LS	979	808	2

ber of function evaluations (error and gradient calculations) and iterations (weight updates) required for the total squared output error to drop below 0.16, for various minimization algorithms. An initial function evaluation performed by **Xerion** on all training runs was ignored for the minimization methods not requiring a line search, as the initial function evaluation is then unnecessary. Averages are for the successful trials only. Number of failures are listed separately. Failures occurred when no solution was found after 5000 iterations (as was the case with momentum, or when training stopped abnormally due to imprecise calculation (as was the case with the conjugate gradient methods). It was decided not to use Theorem 3.1 to incorporate the failed training runs into the averages because when an algorithm went over the allowed "time limit" generally no learning was taking place, meaning W in equation 3.1 could take on any arbitrary value.

To gauge the total computation required by an algorithm, one need only look at the number of function evaluations required. Recall that a function evaluation is the output error and gradient calculation, accumulated over all training patterns when using batch updating. This computation is the same regardless of the minimization method in use. Further computation required only at the end of an iteration, such as the extra computation required by the minimization algorithm and the implemen-

tation of the weight change, can be neglected. It is considerably less than the total of the output error and weight error derivative computations performed after each pattern presentation.

Different minimization methods clearly lead to different learning times for the parity problem. The conjugate gradient methods, when successful, both required considerably less function evaluations than the steepest descent methods. They both required considerably more computation than the momentum and delta-bar-delta methods, however. Steepest descent with a line search was actually slower than ordinary backpropagation because of the large number of function evaluations required by the line search. It appears that complex minimization algorithms with line searches are unnecessary for a small problem such as parity. Simpler algorithms such as momentum and delta-bar-delta achieve considerable reductions in the amount of computation required, and should always be considered for problems such as this.

The type 2 problem was another well-known problem among neural network researchers—classification of sonar signals, originally studied by Gorman and Sejnowski [38]. The task is to train a network to discriminate between sonar signals bounced off a metal cylinder and those bounced off a roughly cylindrical rock, at various aspect angles. There were a total of 208 input-target pattern pairs in the original dataset. Each input pattern contained 60 numbers in the range 0.0 to 1.0, each representing the energy within a particular frequency band, integrated over a certain period of time.

A copy of the original dataset was obtained from Carnegie Mellon University's neural network benchmark collection [39]. The division of the data into training and test sets (104 patterns in each) conformed to Gorman and Sejnowski's method of

Table 3.7: Original results reported for sonar problem

Hidden units	% Right on training set	% Right on test set
0	79.3	73.1
2	96.2	85.7
3	98.1	87.6
6	99.4	89.3
12	99.8	90.4
24	100.0	89.2

generating "aspect-angle dependent "data. Here the division is controlled to ensure that each dataset contains cases from each aspect angle in equal proportions.

Gorman and Sejnowski reported the results shown in Table 3.7 on training and test data from the aspect-angle dependent experiment, for varying numbers of hidden units, averaged over 10 training runs. They used backpropagation with learning rate 2.0 on networks with logistic neurons, and weights initialized in (-0.3, 0.3). Two output neurons were used, one for the "mine" class and one for the "rock" class. Output errors less than 0.2 were treated as zero. They reported a result of 82.7% using a more traditional nearest neighbor classifier.

It was decided that 6 hidden units would be used for the simulations, because no significant improvement in performance was found by Gorman and Sejnowski when more hidden units were used, and to ensure a small enough network that many training runs could be done. Fahlman has noted that the training data are linearly separable, requiring no hidden units for 100% correct classification. However, generalization ability may suffer if a network is trained to 100% accuracy on training data. For the simulations reported here, cross validation was performed with parameters as shown in Table 3.8. The minimization methods that were tried for the parity problem were also tried for this problem. Two methods were also tried using on-line updating:

Table 3.8: Sonar Signal Classifier: Cross Validation Parameters

Minimization method	Iters...between checks	...for improve	Max iters
Ordinary backprop	50	250	2000
Momentum	10	50	500
Delta-bar-delta	10	50	500
Quickprop	10	50	500
Regular CG	5	25	250
Rudi's CG	5	25	250
Steepest descent LS	50	250	1000
On-line bp	1040	5200	52000
On-line mm	520	2600	26000

ordinary backprop, and momentum. Tanh neurons were used in the networks. The ϵ parameter was set to 0.005 for all methods requiring it (this was found to be optimal for both batched *and* on-line backprop).

The results, shown in tables 3.9 and 3.10, were obtained assuming that for correct classification, the output corresponding to the correct class had to have activation greater than that of the other output. No patterns were treated as unclassifiable, to simplify analysis of the results. A weight decay of 0.005 (λ in equation 2.25) was used to generate the results in the second table. All entries in the tables are averaged over 30 training runs. The numbers given for iterations and function evaluations appear high for on-line backprop and on-line momentum because for these each iteration and function evaluation is completed after a single pattern presentation instead of epoch. If these numbers are divided by the number of training patterns (104) one can calculate the number of passes through the training data and compare the on-line algorithms against the others more easily. Note, however, that the on-line algorithms now require slightly more *computation* than the others per pass through the training data, as multiplications by learning rate, momentum parameter, etc. are now being done after each pattern presentation instead of epoch.

Table 3.9: Performance of Various Minimization Methods on Sonar Problem

Minimization method	F.Evals.	Iterations	Train acc.	Test acc.
Ordinary backprop	773	773	91.38	82.53
Momentum	80	80	90.26	85.67
Delta-bar-delta	44	44	81.67	81.35
Quickprop	117	117	82.82	82.95
Regular CG	386	44	87.69	84.49
Rudi's CG	211	35	85.29	84.65
Steepest descent LS	602	437	89.80	83.75
On-line bp	15461	15461	83.56	82.85
On-line mm	4749	4749	81.28	79.39

Table 3.10: Performance of Minimization Methods on Sonar with Weight Decay

Minimization method	F.Evals.	Iterations	Train acc.	Test acc.
Ordinary backprop	870	870	92.47	83.78
Momentum	82	82	88.05	84.74
Delta-bar-delta	46	46	82.08	82.21
Quickprop	105	105	81.44	82.37
Regular CG	290	35	85.48	84.36
Rudi's CG	236	37	86.73	84.94
Steepest descent LS	633	475	89.23	83.75
On-line bp	26936	26936	86.28	84.36
On-line mm	5079	5079	69.49	75.58

The methods that made compromises between previous and present weight error derivatives to calculate a weight change (the two conjugate gradient methods and momentum) seemed to do the best in terms of percent correct classifications on the test data. The momentum and delta-bar-delta algorithms required an order of magnitude less computation (in terms of function evaluations) than ordinary backprop with batch updating. On-line momentum appears to be a poor choice for this problem, based on its poor classification accuracy.

Using weight decay helped the generalization performance of the ordinary backprop (batched and on-line), delta bar delta, and Rudi's conjugate gradient methods only. It is very possible that the weight decay parameter should be adjusted differently for each minimization method (it was only optimized for ordinary backprop) to achieve better results. The same may also be true of the other parameters, although certain methods did better with no tuning than the ordinary backpropagation method, where the optimal learning rate was used.

Average percent correct classification on the training and test sets were consistently lower than the 99.4 and 89.3 % reported by Sejnowski et al. However, it was noted that one training run using batched momentum yielded 92.3% accuracy on the test set, and several others yielded generalization accuracy greater than 89.3%. The cross validation method therefore appears to find good solutions, and saves computation time by giving up when no good solution is being found quickly.

The results presented for parity and sonar signal classification do not attempt to show that one minimization method is better than another in general (although ordinary backprop with batch updating appears quite poor)—much more experimentation would be required for this. They merely try to point out that for a given problem some

Table 3.11: Cross Validation Parameters for Cascade Correlation Sonar Classifier

	Phase of training		
Iterations...	Initial (no hidden units)	Candidate	Output
between validation checks	10	10	1
for generalization improvement	50	40	5
Maximum iterations allowed	500	200	50

Table 3.12: Cascade Correlation Sonar Signal Classifier Results

Hidden units	% Right on training set	% Right on test set
0	86.9	76.1
2	95.8	88.6
3	97.2	88.9
6	98.8	89.3
12	99.2	89.5
24	98.5	89.2

methods will be better than others, and experimentation with different minimization methods, using cross-validation, will yield better results.

An attempt was also made to use cascade correlation networks for the sonar problem. Both output and candidate units were trained using quickprop, with ϵ set to 0.005. Cross validation parameters were as in Table 3.11.

The results in table 3.12, averaged over 10 runs are, as expected from results presented earlier, better than those obtained by Gorman and Sejnowski for low numbers of hidden units, but slightly worse for higher numbers of hidden units. The average generalization performances are better than the ones in Tables 3.9 and 3.10 because it seemed that training went "more smoothly", and the cross validation method did not ever have to give up prematurely. This is probably due to the absence of the moving target problem.

Chapter 4

Hardware Considerations

It has been said that the future of artificial neural networks as a competitive computational technology will be based upon ease of hardware implementation in addition to performance [41]. This chapter will address some issues that should be considered when attempting to implement the algorithms studied in the previous chapter in hardware. The first section will deal with computational and storage requirements of the algorithms, assuming no particular implementation. The second section will analyze the performance of simple backpropagation with on-line weight updating, assuming a fully parallel implementation using analog hardware components.

4.1 Computational and Storage Requirements

When trying to select any type of classifier for a particular problem, small differences in classification accuracy should not be considered all-important. One must also take into consideration memory usage and ease of hardware implementation [42].

Both MLPs and cascade correlation networks may, after training offline, have their forward computation implemented in hardware quite easily. The only memory requirement is weight storage, and the computations required are all multiplications, additions, and possible table lookups (for the sigmoid calculations). If the networks

are implemented such that several portions of the circuits work in parallel, then tremendous speedups over implementations in general-purpose computers (PCs and workstations) can be achieved. Artificial neural networks in general lend themselves very well to parallel implementation, because of their parallel architectures and locality of computation, each neuron and connection only processing signals available nearby. Several organizations have in fact implemented the forward computation of MLPs in parallel hardware.

The group at AT&T Bell Labs mentioned in Section 3.3.2 have recently implemented a trained analog circuit version of their handwritten character recognizer [26] mostly on a chip called ANNA (Analog Neural Network Arithmetic and Logic Unit) [43]. The chip contains 4096 physical synapses (connections), which are time multiplexed to realize the larger handwritten character recognizer. Eight neurons' computations can be processed in parallel. The weights have 6 bits of resolution and the neuron outputs 3 bits. Amazingly, these are adequate resolutions for all but the last (output) layer of the network, which had to be implemented on a DSP chip. About 1000 characters per second could be processed using the ANNA system, compared to about 2 per second on a Sun SPARC 1+ workstation.

Another analog circuit implementation of neural network forward computation is the ETANN (Electrically Trainable Analog Neural Network) from Intel corporation [44]. This chip contains 64 neurons and more than 10,000 modifiable weights. It can process over 2 billion multiply-accumulate operations, or connections, per second, compared to a few million on a Sun SPARC workstation. The weight modification algorithm for ETANN must be done off-chip.

The forward computation of Cascade Correlation networks is slightly more difficult

to implement than that of MLPs because of the networks' non-regular structures, and a fully parallel version will take somewhat longer for classification than a fully parallel MLP because of the large number of layers.

When trying to implement neural network learning computations in hardware, one is faced with the choice of using "general" ANN hardware designed so that many types of neural network algorithms can run on them, or "dedicated" ANN hardware designed for one algorithm only. The advantage of using general hardware is that many types of networks can be experimented with; the advantage of using dedicated hardware is that it is faster and smaller.

Several organizations offer general parallel digital ANN hardware. Examples include Adaptive Solutions' CNAPS (Connected Network of Adaptive ProcessorS) system [45] and the Music system developed at the Swiss Federal Institute of Technology [46]. The CNAPS system uses general-purpose digital parallel processing units in a SIMD[1] fashion and achieves a speedup of up to 1000 times over a Sun SPARCstation when training a network using backpropagation. The Music system uses DSP chips in parallel, and achieves a speedup of about 500 times.

Implementation of neural network learning computations in dedicated hardware is not always so easy. The ordinary backpropagation algorithm, however, consists of simple multiplications and additions, as shown in equations 2.4 and 2.5, and all computation is local, so hardware implementation is not very difficult. The cascade correlation algorithm, on the other hand, would be quite difficult to implement in dedicated hardware. Averages must be calculated and stored, signs of covariances must be calculated and stored, neurons must be added during learning, and there are

[1]Single Instruction, Multiple Data: Each processor at any one time performs the same mathematical operation, but on different data

two alternating phases of training (output unit and candidate unit). It is doubtful that anyone will attempt to implement the full cascade correlation algorithm in dedicated hardware in the near future.

Assuming one tries to implement the ordinary backpropagation algorithm in dedicated hardware, one may also try to implement a minimization algorithm to use with backpropagation. As mentioned in Section 3.3.3, if batch updating is used and the minimization algorithm only operates after the last pattern has been presented, the proportion of time required for the extra computation will be small. What is more important is the *type* of computation involved (some are more difficult to implement than others, for example division is more difficult to implement than addition), and whether all computation can be done locally or must involve accumulations of values present throughout the network, thereby increasing the circuitry required. Table 4.1 is meant to serve as a guide to anyone planning to implement a neural network minimization algorithm in dedicated hardware. Computation types, and storage requirements at each weight are listed. A "dot product" is performed by multiplying two values at each weight and then taking the sum of the results over all weights (a nonlocal computation).

When choosing an algorithm to implement in dedicated hardware, simplicity becomes the prime consideration and one would have to consider implementing plain backpropagation with on-line weight updating. At least one group of authors refused to implement even the momentum algorithm because the extra computation and storage made it too complex in their view [49]. One should only attempt to implement an additional minimization algorithm if it is known that the accuracy of the final solution will be much greater. Training time is of lesser importance in dedicated

Table 4.1: Minimization Algorithm Storage Requirements and Computation Types

Algorithm	Computation type	Extra storage at each weight
Momentum	At weights: +, x	· Previous weight change
Delta-bar-delta	At weights: +, -, x, comparison	· Learning rate · Average of past weight error derivatives (w. e. d. 's)
Quickprop	At weights: +, -, x, /, comparison	· Current w. e. d. · Previous w. e. d. · Past weight change
Regular conjugate gradient	At weights: +, -, x Elsewhere: / Dot products required.	· Current w. e. d. · Previous w. e. d. · Weight's contribution to search direction
Rudi's conjugate gradient	At weights: +, -, x Elsewhere: -, x, / Dot products required.	· Current w. e. d. · Previous w. e. d. · (current - previous) w. e. d. (not essential, but useful) · Previous weight change · Weight's contribution to search direction
Ray's line search	At weights: none Elsewhere: +, -, x, /, square root, comparison Dot products required.	· Possibly previous w. e. d. Search direction and gradient supplied by minimization algorithm.

Add extra storage at each weight for accumulated weight error derivatives if batch updating used.

hardware; any algorithm should work very quickly. Simple backpropagation with on-line updating did not fare too badly on the problems studied in the previous chapter and should be considered for hardware implementation. The next section will do a performance analysis of simple backpropagation with on-line updating, assuming it has been implemented in dedicated analog hardware.

4.2 Analysis of an Analog Hardware Backpropagation Circuit

There has been considerable work on implementing the backpropagation algorithm in dedicated hardware in order to speed up the computation. Many networks use limited-precision digital hardware to perform all or most of the computation; these networks have been analyzed in [47] and [48]. In [49] the forward computation was performed in analog hardware but the remainder of the computation was performed digitally. Analysis usually involves the effect of limited resolution in the computation on learning and/or the network outputs. A typical suggested resolution is 16 bits; this large number of bits results in extensive hardware area.

Analog hardware does not suffer from the same drawbacks as digital hardware. Limited resolution is not an inherent feature of the hardware, and the components typically take up much less area than their digital counterparts. Implementing the entire backpropagation learning algorithm in analog hardware is not difficult. Addition operations in analog hardware are easily performed by summing currents on a wire. Multiplications can be done using analog multipliers such as the wide-range Gilbert multiplier [52]. This type of multiplier was recently employed in the implementation of mean-field networks [53]. It was suggested that an extension of the circuits used in that work might be used to implement the backpropagation learning algorithm. The

other operations involved in backpropagation, including the sigmoid functions and subtractions, may also be implemented using these multipliers, as will be shown in section 4.2.2. To our knowledge, no hardware implementation of the full backpropagation algorithm has been attempted with these circuits. Gilbert multipliers have recently been used to implement only the multiplications in the forward computation, not including the sigmoids, in [56].

However, using analog hardware does have its drawbacks. Different analog components, such as multipliers, will produce different results when presented with the same inputs due to variations in transistor dimensions, etc. between the components. Also, the components perform computations that are only approximations to the corresponding mathematical operations of the backpropagation algorithm, due to non-ideal device characteristics and limited voltage ranges.

It appears that previous authors have mainly refrained from attempting full analog implementation and/or analysis of backpropagation circuits, since their "analog" backpropagation networks typically use analog hardware for the forward computation only, and off-chip, floating-point computation for the rest. This is partly because it is not obvious that non-ideal backward computation can correct for non-ideal forward computation (and vice-versa). Networks with analog forward-only computation and floating-point reverse computation are presented in [50], [51], and [56]. One should attempt to also implement the reverse computation on-chip in order to allow the network to learn new patterns very quickly.

The analog hardware properties that will be analyzed here include variations in multiplier gains and zero offsets (both those fixed by fabrication and those due to noise), function approximations (due to tanh nonlinearities in the multiplier character-

istics and bounded signal values), and weight decay. In previous work component variation has sometimes been modelled by adding noise to the weight changes [50], [57], [58] rather than to the characteristics of components (multipliers) used to compute the weight changes. Component variation modelled at the components themselves was analyzed only for feedforward competitive learning in [56], and was discounted as of minor importance without explanation in the feedforward-only analog circuit of [51]. Approximations to true multiplications by nonlinear 3-transistor multipliers in [51] was taken care of by using special equations off-line to compensate. Weight decay effects have been analyzed before, but using floating-point networks, in [58]. Not surprisingly, given the lack of previous implementations of full backpropagation neural networks in analog hardware, there is also a lack of systematic studies of the effects of analog hardware properties on backpropagation learning.

4.2.1 Circuit Description

A schematic view of a single neuron and synapse of the circuits to be analyzed is shown in Figure 4.1. This circuit is a direct extension of that presented in [53], [54]. The portion of the circuitry used for the forward (output update) computation is enclosed within the dashed border; the remainder is used for the backward (weight update) computation. "U" is the voltage value corresponding to the abstract value of 1.0 (the maximum neuron output).

There are only two different components in the circuit: Wide-range Gilbert multipliers (components marked with X's in Figure 4.1), and simple two-transistor circuits that act as resistors for current-to-voltage conversion (components marked "I/V" in Figure 4.1).

The wide-range Gilbert multiplier works by accepting two differential voltages as

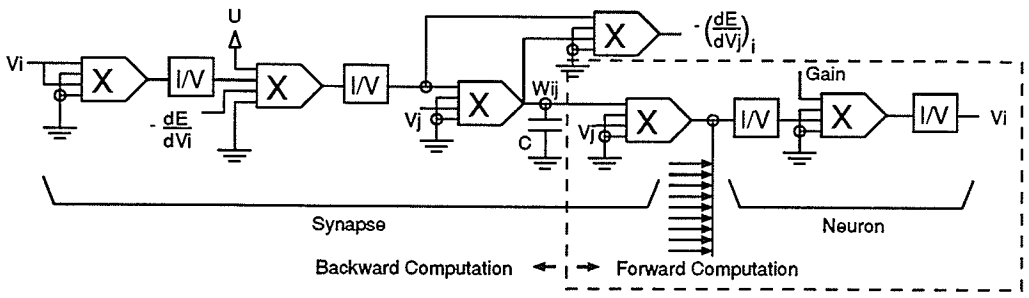

Figure 4.1: Schematic of Analog Circuit for Backpropagation Learning

input, and producing a current as output. The multiplier takes four voltages V_1, V_2, V_3, and V_4 as input and produces a current proportional to $(V_1 - V_2)(V_3 - V_4)$. The inputs to the multipliers in Figure 4.1 correspond, from the top down, to V_1, V_2, V_3, and V_4 respectively. For small- to moderate-sized inputs the current is approximately [53]

$$I_{out} = \sqrt{\frac{\beta^N \beta^P}{2}}(V_1 - V_2)(V_3 - V_4) \tag{4.1}$$

where $\beta = \mu\, C_{ox}$ W/L, the transistor conduction parameter.

Typical characteristics of the Gilbert multiplier are shown in Figure 4.2. Notice that the output current saturates for sufficient input voltage. The characteristics also become nonlinear as this saturation region is approached. This nonlinearity has at least one advantage. The sigmoid nonlinearity of the neuron can be obtained using one of these multipliers, as will be shown in Section 4.2.2. The number of types of components required to build the complete backpropagation circuit is therefore reduced.

The current-to-voltage converter characteristics (not shown) are approximately linear.

Weight changes are performed in the circuit by allowing a charge to flow onto a

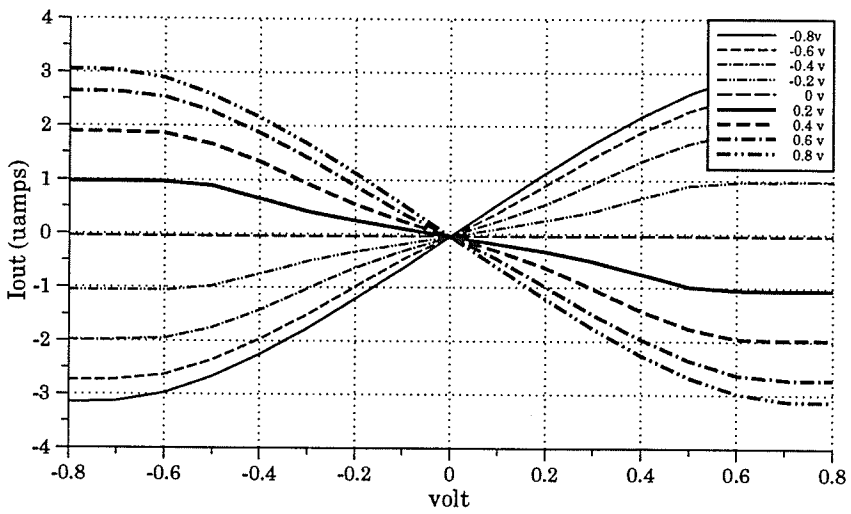

Figure 4.2: Measured Gilbert Multiplier Characteristics
(from Schneider, R. C., "Analog CMOS Circuits for Artificial Neural Networks")

capacitor (whose voltage represents the synaptic weight) for a given length of time t:

$$\Delta w_{ij} = \int \frac{dV}{dt}\, dt = \int (\frac{I}{C})\, dt = \frac{tI}{C} \tag{4.2}$$

where I is the current flowing out of the multiplier with output proportional to $-\partial E/\partial w_{ij}$. Most of the time for learning is due to charging or discharging of the capacitors; very little time is required to compute the weight error derivatives $\partial E/\partial w_{ij}$.

During learning, the multipliers with weights at their outputs must produce a constant current while the weight voltage (the voltage across the capacitor) is changing. As shown in [53], the multipliers perform this function very well over most of their operating range, acting as near-ideal current sources.

Rough estimates of the variations between actual multipliers, which were fabricated using a 1.2μm CMOS process, were $\pm 10\%$ for the gains, $\pm 5\%$ of maximum output for intra-chip zero offset variation, and $\pm 10\%$ of maximum output for inter-

chip zero offset variation[2]. Capacitor leakage current was observed to be very small, on the order of pA. Weight decay is therefore not a problem with these circuits as long as they are presented with patterns from time to time.

4.2.2 Circuit Modelling

At the abstract level, each neuron i outputs the value $out_i = \tanh(net_i)$, where net_i is the net input to neuron i (the weighted sum of inputs, including the bias value). The tanh function was used instead of the logistic function because, from a hardware point of view, it is natural to fabricate circuits with symmetric activation function for positive and negative input voltages. Fortunately, as shown earlier, idealized simulations also dictate the use of tanh neurons. As shown in Section 3.3.3, the tanh function can be used with lower learning rates, meaning a savings in time when using these circuits.

At the circuit level, the equations that generate the characteristic of Figure 4.2 are quite complex (see [53] for details). For simplicity of simulation the multiplier characteristic of figure 4.2 is modelled by the approximation

$$Out = \phi^2 tanh(\sqrt{\theta}\, input_1/\phi)\, tanh(\sqrt{\theta}\, input_2/\phi) + O_y \qquad (4.3)$$

where $input_1$ and $input_2$ correspond to multiplier inputs $(V_1 - V_2)$ and $(V_3 - V_4)$ respectively, ϕ is used to adjust the function range and θ is used to adjust the multiplier's gain. ϕ roughly corresponds to the maximum value that one of the inputs can take before the multiplier becomes very non-linear. The maximum multiplier output is ϕ^2. The term O_y is the zero offset of the multiplier — the actual output when one input is zero. A plot of equation 4.3 for $\phi = 1.0$, $\theta = 1.0$, and $O_y = 0.0$ is shown in Figure 4.3.

[2]zero offset = output produced when one or both inputs are zero

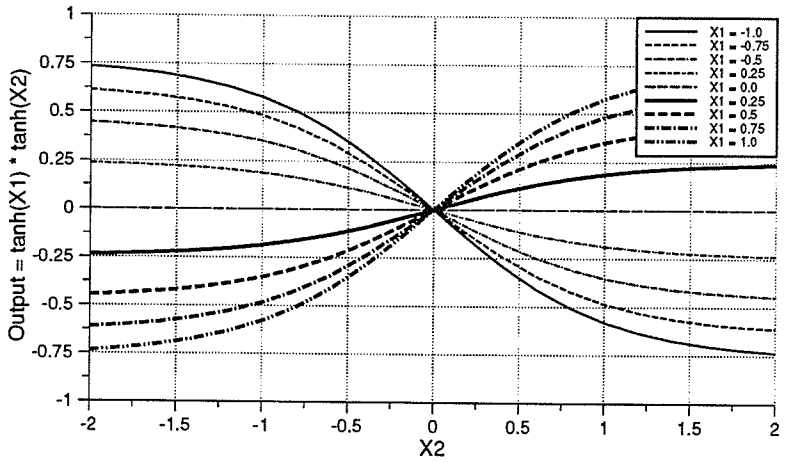

Figure 4.3: Gilbert Multiplier Model: $X1 * X2 => \tanh X1 * \tanh X2$

For all simulations in this section, signal values and parameters such as θ and ϕ are used in abstract terms rather than as voltage values. For example, the neurons have maximum outputs of 1.0 rather than, say, 0.8 volts. This simplifies the simulation process.

Notice that when both inputs are small with respect to ϕ, equation 4.3 becomes

$$Out = \theta\, input_1\, input_2 + O_y \tag{4.4}$$

Equation 4.4 was retained for the multiplier calculating the out_i^2 term of the sigmoid derivative because in this case both multiplier inputs are small (within ± 1.0) with respect to ϕ. The parameter ϕ should obviously be greater than 1.0 to at least allow the network to respond properly throughout the entire range (-2.0 to +2.0) of the desired output minus actual output signal used at the output neurons.

In equation 4.3, if we let $\theta = \phi^2$ (in reality the circuit designer must design the

multiplier, or more simply the I/V converter, to allow for this),

$$Out = \phi^2 \, tanh(input_1) \, tanh(input_2) + O_y \tag{4.5}$$

If $input_2$ is then set to $tanh^{-1}(1/\phi^2)$, then we are left with the familiar hyperbolic tangent activation function. This shows that the multipliers can serve a dual purpose: multiplication and sigmoid calculation.

For the multiply and sigmoid operations, component variability was simulated by varying the gain of the multiplier. By varying the gain of the multiplier we are varying the slope of the multiplier characteristic near the origin. For the multipliers performing the multiply operations this corresponds to varying the θ parameter. For the sigmoids this implies varying the η parameter in the following equation:

$$Out = tanh(\eta \, input_1) + O_y \tag{4.6}$$

It was assumed that the multiplier gains for various circuits distributed over the chip would have a Gaussian-like distribution:

$$gain = (1.0 + devn)^{pow} \tag{4.7}$$

where *pow* is a normal random variable (with Gaussian distribution, mean 0.0 and standard deviation 1.0), and *devn* was a parameter in the simulation, to allow one to observe the degree of component variation the circuits could handle. Possible values for *gain* are from 0.0 to $+\infty$, but for small values of *devn* the gains are distributed roughly symmetrically around 1.0.

Zero offsets were given Gaussian distributions with mean 0.0. Standard deviation of the offsets was another simulation parameter.

4.2.3 Simulation Environment

The simulations were run using the **Xerion** simulator [19] with more enhancements. The backpropagation learning equations were modified to support component variation. Each individual multiply and tanh operation in the network could be assigned its own gain and offset term either at the start of simulation (to simulate variability due to fabrication) or immediately prior to the multiply or tanh operation being required in the computation (to simulate variability due to noise). The means and standard deviations of multiplier gains and offsets were user-settable parameters and a feature was implemented to automatically run a network through a number of simulations, each with progressively greater component variation. The user could also select whether or not the weights and other values corresponding to voltage levels in the circuits should be bounded. This allowed for component variation effects to be studied separately from weight saturation effects. Gaussian random variables for the component variation were generated using code from *Numerical Recipes in C* [33].

When weight saturation was in effect, every multiplication operation was simulated using equation 4.3. The ϕ parameter was made a user-settable parameter, to allow for investigation of the effect of limited voltage range on learning. One could determine, for example, the required range of the multipliers relative to the maximum neuron output.

When saturation was not in effect, the multipliers were simulated using equation 4.4. This corresponds to ϕ (multiplier range parameter) values approaching infinity.

4.2.4 Simulations Performed

A number of different problems were used for simulation, both artificial and real. The problems are the following:

1. The 4-bit parity problem. Same problem as studied in section 3.3.1.

2. The eight-input "hard" overlapping Gaussian problem, the most difficult overlapping Gaussian problem presented in [60]. This problem is expected to be representative of a real-world classification problem, in which the network must discriminate between two classes with Gaussian distributions. For class 1, each input has mean zero and standard deviation one. For class 2, each input has mean zero and standard deviation two. Kohonen reported an error of 18.9% for an 8 hidden unit backpropagation network as compared to the optimal 9.0% achieved by a Gaussian maximum likehihood classifier. Two outputs were used, one for each class.

3. A small version of the grain classification problem studied in Section 3.3.1. The problem adopted here is simply determining whether or not the input corresponds to an amber durum wheat kernel. Preliminary simulations showed that at least 2 hidden units are necessary to solve this problem (4 were actually used in the hardware simulations presented here).

Table 4.2 shows some simulation parameters for all three problems. All were done using on-line updating, as this is the way the circuits are intended to be used, in order to keep the capacitive weights refreshed. A training epoch is as usual defined as a pass through all the training data. Target values in all problems were -0.7 and +0.7. No cross validation was done, as the emphasis in this section is on determining

Table 4.2: Simulation Parameters

Problem	Inputs	Hidden Units	Outputs	Training data points	Test data points	Learning rate ϵ	Training epochs
4-bit parity	4	8	1	16	-	0.01	5000
Gaussians problem	8	8	2	2000	2000	0.02	50
Grain classifier	10	4	1	242	485	0.02	1000

whether the circuits can learn at all, rather than determining exactly how well they can learn.

For the grain classifier network, a classification was considered "correct" if the network's output was greater than 0.0 for an amber durum kernel, or less than 0.0 for a non-amber durum kernel. For the parity network, a stricter performance measure was used because of the relatively high probability with which the network could correctly classify all patterns. This performance measure was one suggested by Scott Fahlman where the classification is considered "correct" only if the network's output is greater than 0.2 for odd parity, or less than -0.2 for even parity [61]. For the overlapping Gaussian classifier, a classification was correct if the output of the neuron corresponding to the correct class was greater than the output of the other neuron.

4.2.5 Results from Learning in Analog Hardware

Multiplier Gain Variation

The first results presented are from simulations of circuits such as those of Figure 4.1, with saturation effects neglected (ϕ in equation 4.3 $= \infty$), and zero offsets also neglected. The standard deviations of multiplier/sigmoid gain (*devn* in equation 4.7) were varied from 0.0 (ideally matched components) to 1.0 in 0.05 increments. Component variation was generated using one of ten different random number streams.

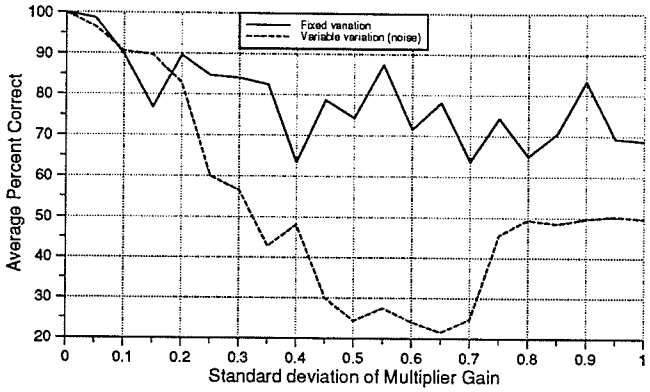

Figure 4.4: Performance of 4-bit Parity Network with Multiplier Gain Variation

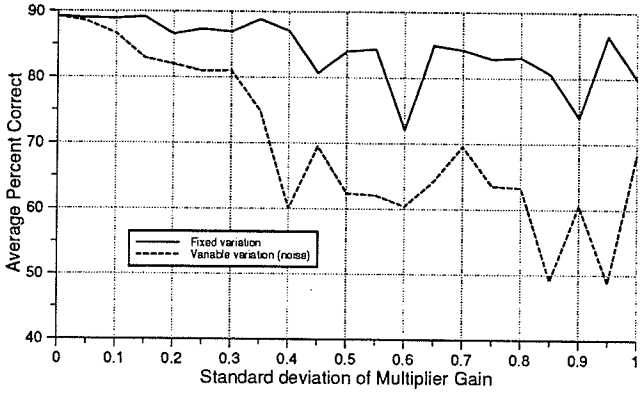

Figure 4.5: Performance of Grain Classifier with Multiplier Gain Variation

Two different sets of starting weights were also employed, yielding a total of twenty simulations for each standard deviation studied. It was desired to distinguish between the effects of noise and of noiseless systematic variation in analog components. Component variations were either fixed at the start of simulation (component variation) or allowed to vary (the case of noise-induced variation). Plots of average percent correct classification (over 20 simulations) for both the fixed and variable cases are shown in Figures 4.4 and 4.5 for the 4-bit parity and grain classification problems respectively. The overlapping Gaussians problem was not attempted without saturation in effect.

The networks performed well when standard deviation of multiplier gain was

within about 20% and 15% of the ideal for the parity and grain problems respectively. These standard deviations are at least as great as the anticipated values. It is clear that on average the networks did better when the gains were fixed at the start of simulation, rather than when they were variable. This is an encouraging result because it shows that the networks are able to make some sense out of a fixed nonideal environment, and can learn to compensate. They exhibit considerably less tolerance to random variation, or noise. Increases in accuracy for high (and probably unrealistic) standard deviations for the parity network occurred because the network tended to classify the patterns as all even or all odd parity with great confidence (outputs very close to -1 or +1), thereby achieving at least 50% accuracy.

Limited Voltage Ranges

Instead of multiplier gain, the ϕ (square root of range) parameter was varied, from 1.0 to $\sqrt{10.0}$ to allow for multiplier ranges from ± 1.0 to ± 10.0. The multiplier gains were maintained at the ideal value (= 1.0). All weights and signals, including summations, were clipped at the maximum multiplier output ϕ^2 because of the voltage limits that would be experienced within an actual circuit. These simulations attempt to determine the required range of the multipliers and weights. Variation in multiplier gain is not expected to have any greater effect when the voltage ranges are limited because for high input values, the multipliers will then produce outputs near the limits of their range regardless of their gains.

Final percent correct classifications, averaged over 4 simulations each with different starting weights, are shown in Figure 4.6 for the 4-bit parity, grain classification, and overlapping Gaussian problems. Average performance with unlimited range

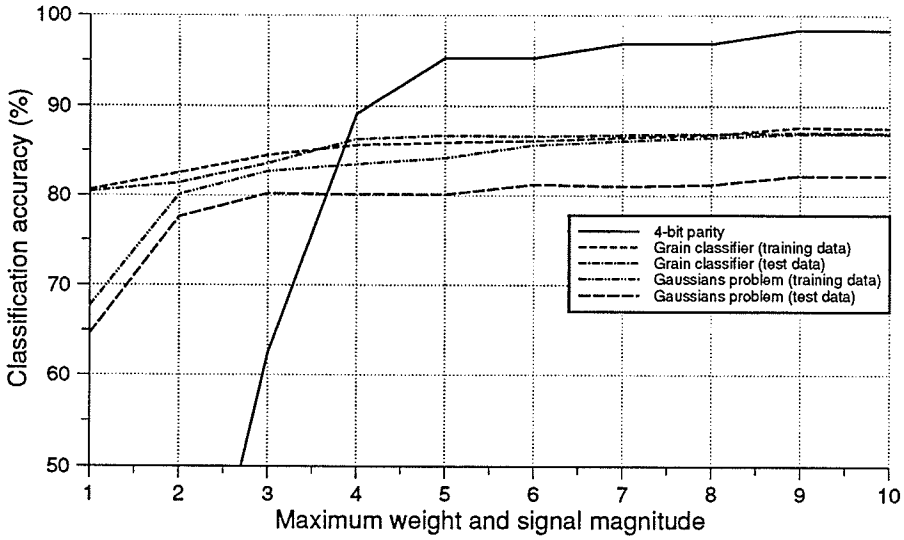

Figure 4.6: Average Classification Accuracies with Limited Voltage Ranges

($\phi = \infty$) was 100% for parity, 88.9 and 87.3% for the grain classifier's training and test data respectively, and 88.4 and 84.2% for the gaussian problem's training and test data respectively.

The networks achieved reasonable classification accuracy relative to that achieved with unlimited range, as long as the ranges of the multipliers and weights were from about -5.0 to +5.0. In other words the voltage value corresponding to unity (1.0) in the circuit should be set to about one-fifth that of the multiplier maximum output, which is not unreasonable given that the multiplier maximum output will be several volts. For networks with larger neuron fan-ins the range may have to be greater to account for larger possible net inputs. Also, the range may have to be greater for networks with many outputs. There may be a problem with the $\partial E/\partial V_j$ term for the hidden neurons in this case, because this term involves a summation that may become quite large for a large number of output units.

Table 4.3: Classification accuracies (%) with weight decay

	Weight decay parameter λ			
Problem	0	0.001	0.01	0.1
4-bit parity	95.31	23.44	0.0	0.0
Grain classifier				
(training data)	85.85	81.92	80.58	80.58
(test data)	86.70	81.29	80.62	80.62
Gaussians "hard" problem				
(training data)	84.24	84.11	75.58	50.00
(test data)	80.05	81.31	73.45	50.00

Weight Decay

Another set of simulations was performed to examine the effect of weight decay on learning, when coupled with saturation effects. This was done because in reality the capacitive charge in the circuits will slowly leak away until it is refreshed by dedicated circuitry [67] or by repeated training patterns [55]. In the presence of weight decay, the weights change according to equation 2.25, losing $\epsilon\lambda$ times their magnitude after each pattern presentation.

In the simulations, λ values of 0.001, 0.01, and 0.1 were used. Generally, the networks were only able to learn anything when $\lambda = 0.001$; even with $\lambda = 0.01$ the networks were unable to learn properly. This implies that between refreshes (pattern presentations) the capacitors cannot afford to lose more than about 0.01% of their charge. Averages of percent correct classification over 4 simulations for the 4-bit parity, grain classifier, and overlapping Gaussian classifier network are shown in Table 4.3 for weight decays of 0, 0.001, 0.01, and 0.1. Each simulation was run using a multiplier maximum output (and weight maximum value) of 5.0.

Table 4.4: Average classification accuracies (%) with zero offsets

	Std.Devn. of Multiplier Additive Offset(% of Max. Output)							
Problem	0.0	0.1	0.2	0.5	1.0	2.0	5.0	10.0
4-bit parity	78.1	96.6	95.9	77.5	77.5	62.2	55.6	49.1
Grain classifier								
(training data)	88.1	84.6	82.9	81.4	80.3	79.8	59.9	62.9
(test data)	87.0	83.5	82.2	80.9	80.5	79.5	59.7	62.8
Gaussians problem								
(training data)	84.4	72.5	60.5	55.5	42.4	44.4	37.6	40.5
(test data)	80.1	68.3	59.3	54.6	41.3	44.4	37.5	40.6

Zero Offset Variation, Gain Variation, and Limited Voltage Ranges

The final results presented here are from simulations conducted assuming limited voltage ranges, gain variation and offset terms added to each multiplication operation. Results for all problems are shown in Table 4.4. A total of 20 simulations were run for the parity and Gaussians problem (2 sets of initial weights and 10 random number streams to generate component variation), and 10 simulations for the grains problem (2 sets of initial weights and 5 random number streams). Each simulation was run using a multiplier maximum output (and weight maximum value) of 5.0. Standard deviation of multiplier gain variation was fixed at 0.15. It is clear that the networks could not learn to overcome the multipliers' additive offsets for degrees of variation near the anticipated values of 5 and 10% of maximum output. This is probably due to the fact that very small weight changes are required by the backpropagation algorithm during learning, and the additive offsets do not permit these small changes. There was one observed case where additive offsets helped, however—the case of 4-bit parity. For some reason, possibly not enough simulation, the 4-bit parity problem could not be learned properly without additive offsets when standard deviation of multiplier gain variation was set at 0.15 (see also Figure 4.4).

Chapter 5

Conclusions and Future Work

The backpropagation and cascade correlation learning algorithms have been analysed with regards to effectiveness on several real-world classification problems, assuming either floating-point computation or (in the case of backpropagation) computation using analog hardware components. The cascade correlation algorithm was successfully programmed into the **Xerion** public domain neural network simulator. Various minimization methods were presented and attempted for use with each of backpropagation and cascade correlation. A method for properly training networks using cross-validation to achieve better generalization performance was presented for use with supervised neural network learning algorithms in general, and was used in many simulations.

Neural networks were found to perform better than a more traditional Gaussian Maximum Likelihood classifier on a real-world classification problem, that of classifying cereal grain kernels. Accuracy on some data not used for training the networks was in general about 3 to 4% greater than accuracy achieved by the Gaussian classifier. The fact that the neural networks made no assumptions about the distribution of the training data was an advantage. It was found on this and other problems that cascade correlation networks could achieve better classification accuracy than

backpropagation-trained networks when a very small sized network (in terms of hidden units) was used, but for larger networks the performance advantage was not present. It would appear that it is very dificult to train all the weights of a small network in an optimal way using backpropagation, and large cascade correlation networks, although training proceeds very smoothly, perform poorly in the end because they seem to commit themselves (by freezing weights) to poor solutions early on during learning.

A handwritten character recognition problem was successfully solved using a hierarchical neural network trained using simple backpropagation. A more elaborate conjugate gradient algorithm used on this network did not yield significantly better classification results. It did however require a longer training time because it required waiting for the entire training dataset to be presented, a waste of time because of the redundancy in the data. The hierarchical nature of the network was found to be the key to solving this problem.

General benchmarking studies on two more problems yielded the result that different minimization algorithms may result in different classification performance, and will certainly result in differences in required computation. These differences in computation may be an order of magnitude. The simplest algorithm to implement in dedicated hardware, ordinary backpropagation with weight updating after each pattern, was shown to perform reasonably well compared to more complex algorithms, and an implementation of simple backpropagation in analog hardware was considered. It was found that the algorithm was tolerant to all the anticipated nonidealities of the hardware, except for the additive offsets between analog multipliers, probably because of the small weight changes that must be guaranteed for the backpropagation

algorithm to work properly.

There is much experimentation one can do with neural networks, and many algorithms to discover. One algorithm in particular should be considered as an alternative to the backpropagation algorithm if one wants to build a small neural network in analog hardware. This is the **weight perturbation** algorithm, which works by tweaking each weight individually, determining the change in output error, doing the same for all other weights, and then adjusting the weights appropriately to lower the error [68].

If one wants to experiment further with backpropagation neural networks, there are many types of neurons and error measures that were not explored in this thesis. Neurons with bell-shaped, linear, or exponential transfer functions can be used, and there are other error measures that work very well for classification problems, such as the soft-max measure. Here each neuron outputs a probability measure that depends on not only its own output, but also the outputs of all other neurons [17].

Cascade correlation networks could be extended by constructing the networks in a hierachical manner, to enable them to solve problems such as handwriting recognition with limited preprocessing where hierarchical networks are required. A fixed hidden layer could be used to combine signals from the neurons that are being added in the "lower" layers of the network hierarchy. Another idea that should be investigated, one suggested in [10], is that of using pools of candidate units with each neuron having a different transfer function. In this way, the network will add neurons of whichever type is appropriate at that particular time. The network designer will then be freed from determining both the appropriate number and appropriate type of neurons, possibly leading to the solution of problems previously unsolvable.

Appendix A

Advanced Algorithms

Ray's Line Search

Much of the following is taken from R. Fletcher,*Practical Methods of Optimization*, Volume 1, Unconstrained Optimization, pp.25ff [21], with some hints from the **Xerion** simulator documentation. The initial guess at the stepsize (given in step 1 below) is **Xerion** 's "slope ratio initial step". There are other initial step methods supplied with **Xerion** , but these are claimed not to work as well.

As in any line search, Ray's includes a **bracket** within which the search must take place. The value λ in equation 2.6 is constrained to lie within this bracket (λ_1,λ_2). The line search algorithm uses **sectioning** to reduce the size of the bracket, and **interpolation** to find an acceptable λ close to the optimal value. Interpolation involves approximating equation 2.6 by fitting a quadratic polynomial in λ to the data, and choosing a new λ that minimizes this polynomial. The algorithm proceeds as follows:

1. Initially, given function (output error) f, slope $s = \mathbf{g} \cdot \mathbf{d}$, where $\mathbf{g}$ is the gradient and $\mathbf{d}$ is the search direction obtained from steepest descent, conjugate gradient, etc., $\lambda_1 = 0, \lambda_2 = \infty, \lambda = \lambda$ s(at beginning of previous line search) / s(current) [unless first step of minimization, in which case $\lambda = 1/ \mid \mathbf{g} \mid$].

2. Evaluate $f^{new} = f(\mathbf{x}+\lambda\mathbf{d})$, where $\mathbf{x}$ is the weight vector. **Xerion** also evaluates $\mathbf{g}^{new} = \mathbf{g}(\mathbf{x} + \lambda\mathbf{d})$, and $s^{new} = \mathbf{g}^{new}\,\mathbf{d}$ at this point.

3. Check whether the function value has decreased by an acceptable amount:

$$f - f^{new} \geq -\rho\,\mathbf{g}\,\mathbf{d}\,\lambda$$

4. If check succeeded, then goto 5.
   Otherwise, calculate a new λ using restricted quadratic interpolation:

$$\lambda(new) = \frac{\lambda - \lambda_1}{2 + 2\frac{f - f(new)}{(\lambda - \lambda_1)s}} + \lambda_1.$$

Restrict λ^{new} to lie within $[\lambda_1 + \tau(\lambda_2 - \lambda_1), \lambda_2 - \tau(\lambda_2 - \lambda_1)]$.
Then set $\lambda_2 = \lambda, \lambda = \lambda^{new}$, and goto 2.

5. Check whether the slope has decreased by an acceptable amount:

$$\left|\frac{s^{new}}{s}\right| \leq \sigma$$

This is sometimes called the **Wolfe** test.

6. If check succeeded, we're done.
Otherwise, calculate a new λ using restricted quadratic extrapolation:

$$\lambda^{new} = \frac{(\lambda - \lambda_1)s^{new}}{(s - s^{new})} + \lambda$$

It may be necessary to use $\lambda^{new} = 2$, if the above calculation fails.
Restrict λ^{new} to lie within $[\lambda_1 + \tau(\lambda_2 - \lambda_1), \lambda_2 - \tau(\lambda_2 - \lambda_1)]$.
Also, restrict $\lambda^{new} - \lambda_1 \leq \eta(\lambda^{new} - \lambda_1)$.
Then set $\lambda_1 = \lambda, f = f^{new}, s = s^{new}, \lambda = \lambda^{new}$, and goto 2.

Rudi's Conjugate Gradient

Adapted from the **Xerion** neural network simulator manual:
Rudi's conjugate gradient computes the following:

$$\begin{aligned}
u1 &= \frac{\mathbf{s} \cdot \nabla E^{new}}{(\nabla E^{new} - \nabla E^{old}) \cdot (\nabla E^{new} - \nabla E^{old})} \\
u2 &= \frac{(\nabla E^{new} - \nabla E^{old}) \cdot \nabla E^{new}}{(\nabla E^{new} - \nabla E^{old}) \cdot (\nabla E^{new} - \nabla E^{old})} - \frac{2\,\mathbf{s} \cdot \nabla E^{new}}{(\nabla E^{new} - \nabla E^{old}) \cdot \mathbf{s}} \\
u3 &= \frac{\mathbf{s} \cdot \nabla E^{new}}{(\nabla E^{new} - \nabla E^{old}) \cdot (\nabla E^{new} - \nabla E^{old})} \\
\mathbf{s}' &= u1 * (\nabla E^{new} - \nabla E^{old}) + u2 * \mathbf{s} - u3 * \nabla E^{new}
\end{aligned}$$

where
s' is the new search direction,
s is the step taken in the previous search direction (vector of previous weight changes),
∇E^{new} is the current gradient (vector of current weight error derivatives), and
∇E^{old} is the previous gradient (vector of previous w.e.d's).

The Pseudo-Newton Method

For every connection between two neurons i and j, where neuron i is closest to the output:

$$\Delta w_{ij} = -\frac{\partial E}{\partial w_{ij}} / \frac{\partial^2 E}{\partial w_{ij}^2}$$

where $\partial E/\partial w_{ij}$ is calculated using one of the backpropagation equations 2.4 or 2.5 depending on whether neuron i is an output or hidden neuron, and

$$\frac{\partial^2 E}{\partial w_{ij}^2} = V_j^2 f'(net_i)[f''(net_i)(V_i - V_i^d) + f'(net_i)] \tag{A.1}$$

if neuron i is an output neuron. V_i is neuron i's output. $f'(net_i) = \partial V_i/\partial X_i$, where X_i is the net input to neuron i (i.e. $f'(net_i)$ is the derivative of neuron i's activation function).

If neuron i is a hidden neuron,

$$\frac{\partial^2 E}{\partial w_{ij}^2} = f''(net_i)f'(net_i)V_j^2 \sum_u \frac{\partial E}{\partial X_u} w_{ui} + [f'(net_i)]^2 V_j^2 \sum_u \sum_v w_{ui} w_{vi} \frac{\partial^2 E}{\partial X_u \partial X_v} \tag{A.2}$$

where u and v are neurons in the next "higher" layer (closer to the output) than neuron i.

If u is an output neuron, then

$$\frac{\partial E}{\partial X_u} = (V_u - V_u^d) f'(net_u) \tag{A.3}$$

and

$$\frac{\partial^2 E}{\partial X_u^2} = [f'(net_u)]^2 + (V_u - V_u^d) f''(net_u^2) f'(net_u). \tag{A.4}$$

If u is a hidden neuron, then

$$\frac{\partial E}{\partial X_u} = f'(net_u) \sum_y \frac{\partial E}{\partial X_y} w_{yu} \tag{A.5}$$

and

$$\frac{\partial^2 E}{\partial X_u^2} = f''(net_u)f'(net_u) \sum_y \frac{\partial E}{\partial X_y} w_{yu} + [f'(net_u)]^2 \sum_y \sum_z w_{yu} w_{zu} \frac{\partial^2 E}{\partial X_y \partial X_z} \tag{A.6}$$

where y and z are neurons in the next higher layer.

If y and z are two distinct output neurons, then

$$\frac{\partial^2 E}{\partial X_y \partial X_z} = 0, \tag{A.7}$$

and if they are two distinct hidden neurons, then

$$\frac{\partial^2 E}{\partial X_y \partial X_z} = f'(net_y)f'(net_z) \sum_a \sum_b w_{az} w_{bz} \frac{\partial^2 E}{\partial X_a \partial X_b}, \tag{A.8}$$

where a and b are neurons in a still higher layer.

Notice that whenever the final equation must be used, which is when there are at least three layers in the network (one output and two hidden), then the network must perform nonlocal computation, making hardware implementation of the algorithm difficult. The nonlocal computation is required because computation and storage of the $\partial^2 E/\partial X_y \partial X_z$ terms must take place at a location external to the network.

The Full Quickprop Algorithm

The original equation proposed for Quickprop was

$$\Delta w_{ij}(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w_{ij}(t-1) \tag{A.9}$$

where $S(t)$ and $S(t-1)$ are the current and previous values of $\partial E/\partial w_{ij}$. Modified versions of this equation are sometimes used to get the algorithm to work properly. There are three different actions to take depending on which of these conditions are met:

1. S(t), S(t-1) same sign and $|S(t)| < |S(t-1)|$:
   $\Delta w_{ij}(t) = -\epsilon S(t) + \frac{S(t)}{S(t-1) - S(t)} \Delta w_{ij}(t-1)$
2. S(t), S(t-1) different signs:
   $\Delta w_{ij}(t)$ as in equation A.9.
3. S(t), S(t-1) same sign and $|S(t)| > |S(t-1)|$:
   $\Delta w_{ij}(t) = -\epsilon S(t) + \mu \Delta w_{ij}(t-1)$

The parameter μ is known as the "maximum growth factor". It is also used in cases 1 and 2: For case 1, if $S(t)/(S(t-1) - S(t)) > \mu$ then the weight update is made as for case 3. For case 2, if $S(t)/(S(t-1) - S(t)) < -\mu$ then $\Delta w(t) = -\mu \Delta w(t-1)$.

The Gaussian Maximum Likelihood Classifier

The Gaussian maximum likelihood classifier assumes that the input classes have multivariate normal densities, where the probability that an input vector $\mathbf{x}$ belongs to class w_i is

$$p(\mathbf{x}|w_i) = \frac{1}{(2\pi)^{n/2}|C_i|^{1/2}} \; e^{-\frac{1}{2}(\mathbf{x}-\mathbf{u}_i)^T C_i^{-1}(\mathbf{x}-\mathbf{u}_i)} \tag{A.10}$$

where n is the number of input features, $\mathbf{u}_i$ is the estimated class mean vector, and C_i is the estimated class covariance matrix [30]. $\mathbf{x}$ is classified as belonging to w_i if

$$D_i(\mathbf{x}) > D_j(\mathbf{x}) \qquad \forall_{j \neq i} \tag{A.11}$$

where each discriminant function $D_i(\mathbf{x})$ is given in terms of equation (A.10) and the *a priori* probability of belonging to class w_i, $P(w_i)$, by

$$D_i(\mathbf{x}) = ln(P(w_i)\; p(\mathbf{x}|w_i))$$

$$= -\frac{n}{2} ln(2\pi) \; - \; \frac{n}{2} ln|C_i| \; - \; \frac{1}{2}(\mathbf{x} - \mathbf{u}_i)^T C_i^{-1}(\mathbf{x} - \mathbf{u}_i) \; + \; ln(P(w_i). \tag{A.12}$$

This approach yields the theoretically optimal classification accuracy if the input classes truly have multivariate normal densities.

Appendix B

Xerion Parameter Defaults

Before the default parameters for minimization are listed, it must be stated that the two "extension" parameters supplied, namely `zeroErrorRadius` (subtracted from an output neuron's error; result used as error unless negative in which case zero is used), and `weightCost` (λ in equation 2.25) were both set to zero, except for the weight decay experiments where `weightCost` was used as the weight decay parameter.

Learning rate parameter

This is the ϵ term appearing in the equations for ordinary backprop (equation 2.2), momentum (equation 2.9), delta-bar-delta (used as $\epsilon(0)$ in equation 2.13), and quickprop (equation 2.20). For ordinary backprop, momentum, and delta-bar-delta, this term is set using **Xerion** 's `epsilon` parameter, which for some strange reason defaults to 1.2345e-10. This default value was *never* used and was always found through experimentation. For quickprop, `epsilon` must always be set to 1.0; a different **Xerion** parameter corresponds to ϵ in equation 2.20. This is the `qpEpsilon` parameter, which also defaults to 1.2345e-10 but was found through experimentation. This experimentation might have been done by finding an optimal value for the `epsilon` parameter for ordinary backprop, and using this value for `qpEpsilon` when running quickprop.

Momentum

The learning rate setting (ϵ in equation 2.9) is discussed under "Learning rate parameter".

The α parameter appearing in the equations for momentum (equation 2.9) and delta-bar-delta (equation 2.15) is controlled by **Xerion** 's `alpha` parameter, which defaults to 0.9.

Delta-Bar-Delta

The initial learning rate setting ($\epsilon(0)$ in equation 2.13) is discussed under "Learning rate parameter".

The momentum setting (α in equation 2.15) is discussed under "Momentum".

The additive gain increment, κ in equation 2.13, is controlled by `gainIncrement`, which defaults to 0.1.

The multiplicative gain decrement, ϕ in equation 2.13, is controlled by `gainDecrement`, which defaults to 0.9.

Quickprop

The learning rate setting (ϵ in equation 2.20) is discussed under "Learning rate parameter".

The maximum permissable increase in a weight change, μ in the explanation of the full quickprop algorithm in Appendix A, is controlled by the `maxGrowthFactor` parameter, which defaults to 1.75.

Regular Conjugate Gradient

No parameters.

Rudi's Conjugate Gradient

No parameters.

Ray's Line Search

The description of these parameters makes reference to the algorithm description given in Appendix A.

The maximum acceptable slope ratio, σ in step 5, is controlled by the `maxSlopeRatio` parameter, which defaults to 0.5.

The Wolfe test (step 5 of the description in Appendix A) is used by default (the `wolfeTest` parameter is set to 1). Without this test, the absolute value signs in step 5 disappear and any point with a positive slope passes the slope ratio test (assuming the previous slope was negative).

The minimum acceptable function reduction factor, ρ in step 3, is controlled by `minFuncReductionFactor`, which defaults to 0.01.

The parameter controlling how close a new point can be to the ends of an extrapolation interval (τ in step 6) is controlled by `extrapLimits` which defaults to 0.35.

The parameter controlling how close a new point can be to the ends of an interpolation interval (τ in step 4) is controlled by `interpLimits` which defaults to 0.05.

Not mentioned in the algorithm description is a parameter used to control the amount of backtracking when too large a step has been taken. In this case, λ is set to $\lambda_1 + (\lambda - \lambda_1)/\gamma$, where γ is controlled by `backupFactor` which defaults to 3.

The upper bound on step size, λ_2 is not initially set to ∞ but is set using `stepBound`, which defaults to 10.

The η term in step 6 is controlled by `maxExtrapol`, which defaults to 9.

When the λ calculation in step 6 fails, λ is assigned `extensor`, which defaults to 2.

It was not mentioned in the main thesis document that there is a parameter controlling the maximum number of function evaluations in a line search. This parameter is `lsMaxFuncEvals` and defaults to 10. However, throughout the thesis work this parameter was set high enough so that there were never more function evaluations on any line search (which would have halted the entire minimization process).

There are other parameters not listed here that if changed from their defaults would change the line search algorithm presented in Appendix A slightly.

Bibliography

[1] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning Internal Representations by Error Propagation" in *Parallel Distributed Processing: Explorations in the MicroStructure of Cognition*, Vol. 1, D. E. Rumelhart and J. L. McClelland, eds. Cambridge, MA: M.I.T. Press, 1986.

[2] Diehl, S. and Eglowstein, H., "Tame the Paper Tiger", *Byte*, April 1991, 220–241.

[3] Kohonen, T., "The 'Neural' Phonetic Typewriter", *Computer*, March 1988.

[4] Lippman, R. P., "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, April 1987.

[5] Hinton, G. E., "Connectionist Learning Procedures", *Artificial Intelligence*, Vol. 40, 185–234.

[6] Hertz, J., Krogh, A., and Palmer, R. G., "Introduction to the Theory of Neural Computation", Addison-Wesley Publishing Company, Redwood City, CA, 1991.

[7] Bolt, G., "Investigating Fault Tolerance in Artificial Neural Networks", Technical Report YCS 154, Department of Computer Science, University of York (1991).

[8] Sequin, C. H., and Clay, R. D., "Fault Tolerance in Artificial Neural Networks", *Proceeding of the 1990 International Joint Conference on Neural Networks, San Diego*, Vol. 1, 703–708.

[9] Horn, B. K. P., "Robot Vision", The MIT Press, 1986.

[10] Fahlman, S. E. and Lebiere, C., "The Cascade-Correlation Learning Architecture", *Advances in Neural Information Processing Systems 2*, Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990, 524–532.

[11] Pomerleau, D. A., "Alvinn: An Autonomous Land Vehicle in a Neural Network", Technical Report 1140, MIT AI Laboratory , 1989.

[12] Weigend, A. S., Huberman, B. A., and Rumelhart, D. E., "Predicting the Future: A Connectionist Approach", Technical Report SSL-90-20, Palo Alto Research Center , 1990.

[13] Bryson, A. E., and Ho, Y.-C., *Applied Optimal Control*, Blaisdell Publishers, New York, 1969.

[14] Werbos, P., "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences", Ph.D. Thesis, Harvard University, 1974.

[15] Parker, D. B., "Learning Logic", Technical Report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA., 1985.

[16] Minsky, M. L., and Papert, S. A., *Perceptrons*, MIT Press, Cambridge, 1969.

[17] Hinton, G. E., "Neural Networks for Industry", Lecture Notes, 1991.

[18] Hush, D. R., and Salas, J. M., "Improving the Learning Rate of Backpropagation with the Gradient Reuse Algorithm", *Proceedings of the 1989 IEEE International Conference on Neural Networks*, Vol. 1, 441–447.

[19] Van Camp, D., Plate, T., and Hinton, G. E., "The **Xerion** Neural Network Simulator", Department of Computer Science, The University of Toronto, 1991. Available by anonymous ftp from `ftp.cs.toronto.edu` (128.100.1.105) in directory `pub/xerion`.

[20] **Xerion** Neural Network Simulator documentation.

[21] Fletcher, R., "Practical Methods of Optimization", Vol. 1, John Wiley and Sons, Toronto, 1980, 25ff.

[22] Watrous, R. L., "Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization", *Proceedings of the 1987 IEEE International Conference on Neural Networks*, Vol. 2, 619–627.

[23] Jacobs, R. A., "Increased Rates of Convergence Through Learning Rate Adaptation", *Neural Networks*, Vol. 1, 1988, 295–307.

[24] Ricotti, L. P., Ragazzini, S., and Martinelli, G., "Learning of Word Stress in a Sub-Optimal Second Order Back-Propagation Neural Network", *Proceedings of the 1988 IEEE International Conference on Neural Networks*, Vol. 1, 355–361.

[25] Le Cun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D., "Backpropagation Applied to Handwritten Zip Code Recognition", *Neural Computation*, Vol. 1, 1989, 541–551.

[26] Le Cun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D., "Handwritten Digit Recognition with a Backpropagation Network" *Advances in Neural Information Processing Systems 2*, Morgan Kaufman Publishers, Inc., Palo Alto, CA, 1990, 396–404.

[27] Source code for cascade correlation algorithm, available by anonymous ftp from `pt.cs.cmu.edu` (128.2.254.155) in directory `/afs/cs/project/connect/code`.

[28] Sjogaard, S., "A Conceptual Approach to Generalization in Dynamic Neural Networks", Ph.D. thesis, Aarhus University, Denmark, 1991.

[29] Baum, E. B., and Haussler, D., "What Size Net Gives Good Generalization?", *Neural Computation*, Vol. 1, 1989, 151–160.

[30] Fukunaga, K., "Statistical Pattern Classification", in *Handbook of Pattern Recognition and Image Processing*, Tzay Y. Young and King-Sun Fu (eds.), Academic Press, New York, 1986.

[31] Lang, K. J., Waibel, A. H., and Hinton, G. E., "A Time-Delay Neural Network Architecture for Isolated Word Recognition", *Neural Networks*, Vol. 3, 1990, 23–43.

[32] Morgan, N., and Bourlard, H., "Generalization and Parameter Estimation in Feedforward Nets: Some Experiments", *Advances in Neural Information Processing Systems 2*, Morgan Kaufman Publishers, Inc., Palo Alto, CA, 1990, 630–637.

[33] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes in C*, Cambridge University Press, New York, 1988.

[34] Kimura, F., and Shridhar, M., "Handwritten Numerical Recognition Based on Multiple Algorithms", *Pattern Recognition*, Vol. 24, No. 10, 1991, 969–983.

[35] Cao, J., Kimura, F., Shridhar, M., and Ahmadi, M., "Statistical and Neural Classification of Handwritten Numerals: A Comparative Study", submitted to to the 11th IAPR International Conference on Pattern Recognition, The Hague, August, 1992.

[36] Le Cun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., Hubbard, W., "Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning", *IEEE Communications Magazine*, November 1989, 41–46.

[37] Knerr, S., Personnaz., L., and Dreyfus, G., "A New Approach to the Design of Neural Network Classifiers and its Application to the Automatic Recognition of Handwritten Digits", *Proceedings of the 1991 International Joint Conference on Neural Networks, Seattle*, Vol. 1, 91–96.

[38] Gorman, R. P., and Sejnowski, T. J.,"Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets" *Neural Networks*, Vol. 1, 1988, 75–89.

[39] The Carnegie Mellon University Neural Network Benchmark Collection, available by anonymous ftp from `pt.cs.cmu.edu` (128.2.254.155) in directory `/afs/cs/project/connect/bench`.

[40] Hoehfeld, M., and Fahlman, S. E., "Learning with Limited Numerical Precision Using the Cascade-Correlation Algorithm", Technical Report CMU-CS-91-130, School of Computer Science, Carnegie Mellon University, 1991.

[41] Graf, H. P., and Jackel, L. D., "Analog Electronic Neural Network Circuits", *IEEE Circuits and Devices Magazine*, Vol. 5, No. 4, 1989, 44–55.

[42] Ng, K., and Lippmann, R. P., "A Comparative Study of the Practical Characteristics of Neural Network and Conventional Pattern Classifiers", *Advances in Neural Information Processing Systems 3*, Morgan Kaufman Publishers, Inc., Palo Alto, CA, 1991, 970–976.

[43] Sackinger, E., Boser, B. E., Bromley, J., LeCun, Y., and Jackel, L. D., "Application of the ANNA Neural Network Chip to High-Speed Character Recognition", *IEEE Transactions on Neural Networks*, Vol. 3, No. 3, May 1992, 498–505.

[44] ETANN product literature, available from Intel Corporation, 2250 Mission College Blvd., Santa Clara, CA 95052-8125.

[45] CNAPS product literature, available from Adaptive Solutions, Inc., 1400 N.W. Compton Drive, Suite 340, Beaverton, OR 97006.

[46] Muller, U. A., Baumie, B., Kohler, P., Gunzinger, A., and Guggenbuhl, W., "Achieving Supercomputer Performance for Neural Network Simulation with an Array of Digital Signal Processors", *IEEE Micro*, October 1992, 55–65.

[47] Holt, J. L., and Hwang, J.-N., "Finite Precision Error Analysis of Neural Network Electronic Hardware Implementations", *Proceedings of the 1991 International Joint Conference on Neural Networks, Seattle*, Vol. 1, 519–525.

[48] Holt, J. L., and Baker, T. E., "Back Propagation Simulations using Limited Precision Calculations", *Proceedings of the 1991 International Joint Conference on Neural Networks, Seattle*, Vol. 2, 121–126.

[49] Hollis, P. W., Harper, J. S., and Paulos, J. S., "The Effects of Precision Constraints in a Backpropagation Learning Network", *Neural Computation* 2, 1990, 363–373.

[50] Frye, R. C., Rietman, E. A., and Wong, C. C., "Back-Propagation Learning and Nonidealities in Analog Neural Network Hardware", *IEEE Transactions on Neural Networks*, Vol. 2, No. 1, January 1991, 110–117.

[51] Lont, J. B., and Guggenbühl, W., "Analog CMOS Implementation of a Multilayer Perceptron with Nonlinear Synapses", *IEEE Transactions on Neural Networks*, Vol. 3, No. 3, May 1992, 457–465.

[52] Gilbert, B., "A high-performance monolithic multiplier using active feedback", *IEEE Journal of Solid-state Circuits*, 1974, SC-9, 364–373.

[53] Schneider, C. R., "Analog CMOS Circuits for Artificial Neural Networks", PhD thesis, Department of Electrical and Computer Engineering, The University of Manitoba, 1991.

[54] Schneider, C. R., and Card, H. C., "CMOS Implementation of Analog Hebbian Synaptic Learning Circuits", *Proceedings of the 1991 International Joint Conference on Neural Networks, Seattle*, Vol. 1, 437–442.

[55] Schneider, C. R., and Card, H. C., "Analog CMOS Contrastive Hebbian Networks", SPIE Proc. 1709, *Applications of Artificial Neural Networks III*, Orlando, Florida, Apr 21–24, 1992, paper #81, in press.

[56] Choi, J., and Sheu, B. J., "VLSI Design of Compact and High-Precision Analog Neural Network Processors", *Proceedings of the 1992 International Joint Conference on Neural Networks, Baltimore*, Vol. 2, 637–641.

[57] Murray, A. F., "Multilayer Perceptron Learning Optimized for On-Chip Implementation: A Noise-Robust System", *Neural Computation* 4, 1992, 366–381

[58] Mundie, D. B., and Massengil, L. W., "Weight Decay and Resolution Effects in Feedforward Artificial Neural Networks", *IEEE Transactions on Neural Networks*, Vol. 2, No. 1, January 1991, 168–170.

[59] Dolenko, B. K., Card, H. C., Neuman, M., and Shwedyk, E., "Classifying Cereal Grains Using Backpropagation Networks", University of Manitoba Department of Electrical and Computer Engineering TR 91-101.

[60] Kohonen, T., Barna, G., and Chrisley, R., "Statistical Pattern Recognition with Neural Networks: Benchmarking Studies", *Proceedings of the 2nd IEEE International Conference on Neural Networks*, Vol. I, 61–68.

[61] Fahlman, S. E., "An Empirical Study of Learning Speed in Back-Propagation Networks", Carnegie-Mellon University Technical Report CMU-CS-88-162, 1988.

[62] Krogh, A., and Hertz, J. A., "A Simple Weight Decay Can Improve Generalization", University of California at Santa Cruz, preprint.

[63] Sejnowski, T. J., and Rosenberg, C. R., "Parallel Networks that Learn to Pronounce English Text", *Complex Systems*, 1:145–168, 1987.

[64] Kolen, J. F., and Pollack, J. B., "Back Propagation is Sensitive to Initial Conditions", *Complex Systems*, Vol.4 No.3, June 1990, 269–280.

[65] Dolenko, B. K., "Pattern Classification using Backpropagation Neural Networks", BSc thesis, Department of Electrical and Computer Engineering, The University of Manitoba, 1990

[66] Schwartz, D. B., Howard, R. E., and Hubbard, W. E., "A Programmable Analog Neural Network Chip", *IEEE Journal of Solid State Circuits*, Vol. 24, 313–319, Apr 1989.

[67] Hochet, B., Peiris, V., Abdo, S., and Declercq, M. J., "Implementation of a Learning Kohonen Neuron based on a Multilevel Storage Technique", *IEEE Journal of Solid State Circuits*, Vol. 26, 262–267, Mar 1991.

[68] Widrow, B., Lehr, M., and Wan, E., "MRIII: A Robust Algorithm for Training Analog Neural Networks", *Proceedings of the 1990 International Joint Conference on Neural Networks, Washington, D.C.*, Vol. 1, 533–536.